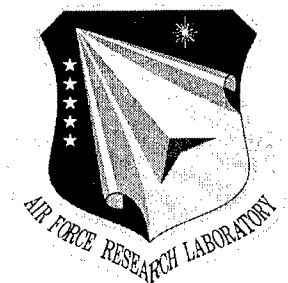


AFRL-IF-RS-TR-2001-161

Final Technical Report

August 2001



CONTEXT SWITCHING RECONFIGURABLE COMPUTING

Sanders, A Lockheed Martin Company

**Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. E223**

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.


20020308 044

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

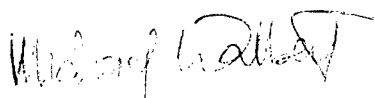
This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2001-161 has been reviewed and is approved for publication.

APPROVED:


MARTIN WALTER
Project Engineer

FOR THE DIRECTOR:


MICHAEL L. TALBERT, Technical Advisor
Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTC, 26 Electronic Pky, Rome, NY 13441-4514. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

CONTEXT SWITCHING RECONFIGURABLE COMPUTING

Stephen M. Scalera

Contractor: Sanders, A Lockheed Martin Company.

Contract Number: F30602-96-C-0350

Effective Date of Contract: 26 September 1996

Contract Expiration Date: 26 June 2000

Short Title of Work: Context Switching Reconfigurable Computing

Period of Work Covered: Sep 96 – Jun 00

Principal Investigator: Stephen M. Scalera

Phone: (603) 885-4029

AFRL Project Engineer: Martin J. Walter

Phone: (315) 330-4102

Approved for public release; distribution unlimited.

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Martin J. Walter, AFRL/IFTC, 26 Electronic Pky, Rome, NY.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
<small>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.</small>				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE Aug 01		3. REPORT TYPE AND DATES COVERED Final Sep 96 - Jun 00
4. TITLE AND SUBTITLE CONTEXT SWITCHING RECONFIGURABLE COMPUTING			5. FUNDING NUMBERS C - F30602-96-C-0350 PE - 62301E PR - E223 TA - 00 WU - P1	
6. AUTHOR(S) Stephen M. Scalera				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Sanders, A Lockheed Martin Company PO Box 868 Nashua, NH 03061-0868			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington, VA 22203-1714 AFRL/IFTC 26 Electronic Pky Rome, NY 13441-4514			10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2001-161	
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Martin J. Walter, IFTC, 315-330-4102				
12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This report describes the development of a new field programmable gate array (FPGA) device that enables dynamic reconfiguration, the changing of hardware logic during normal system operation. The Context Switching Reconfigurable (CSRC) FPGA is capable of storing four configurations on-chip and switching between them on a single clock cycle basis. Configurations can be loaded while other contexts are active, and a powerful cross-context data sharing mechanism has been implemented. This feature allows data to be saved on the device while other programs (context) may operate on the data. This report provides the details of the two-phase development of the CSRC device. The first phase involved the development of a small prototype integrated circuit (IC) version of the CSRC technology. This IC served both as a concept validation tool and platform for acquiring empirical data about the performance enhancements afforded by this new technology. The subsequent phase entailed the development and fabrication of a large IC (greater capacity) with several additional features. Both the prototype and the larger more capable final device are full custom IC designs designated and fabricated on National Semiconductor's .35u line.				
14. SUBJECT TERMS Reconfigurable Computing Module, "logic" configuration, context, performance enhancement, routing, data sharing			15. NUMBER OF PAGES 142	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Abstract

Sanders' Context Switching Reconfigurable Computing (CSRC) program, sponsored by the Information Technology Office (ITO) of the Defense Advanced Research Projects Agency (DARPA), has successfully developed a revolutionary new device that surpasses commercially available field programmable gate arrays (FPGAs). Dynamic reconfiguration of FPGAs, the changing of hardware logic during normal system operation, has recently emerged as the next step in reconfigurable computing. Contemporary FPGA's can at best only partially change portions of the logic during operation, and take too long to reconfigure the logic. In order to take advantage of dynamic reconfiguration, devices must be able to change their configuration as fast as possible. Sanders, A Lockheed Martin Company, under this contract developed the enabling technology to exploit dynamic reconfiguration

The Sanders CSRC FPGA is capable of storing four configurations on-chip and switching between them on a single clock cycle basis. This is at least an order of magnitude faster than conventional FPGAs. Configurations can be loaded while other contexts are active, and a powerful cross-context data sharing mechanism has been implemented. This feature allows data to be saved on the device while other programs(contexts) may operate on the data, truly an innovation in FPGA type devices.

This report provides the details of the two phase development of the Sanders CSRC device. The first phase involved the development of a small prototype integrated circuit (IC) version of the CSRC technology. This IC is the world's first context switchable FPGA and served both as a concept validation tool and a platform for acquiring empirical data about the performance enhancements afforded by this new technology. The subsequent phase entailed the development and fabrication of a larger IC (greater capacity) with several additional features. Both the prototype and the larger more capable final device are full custom IC designs designed and fabricated on National Semiconductor's .35 μ line.

In order to gain wide acceptance of this new technology, design techniques and tools that meet or exceed current expectations of system programmers were provided to the design community and a commercialization path for the CSRC technology was established. The utility of the technology was shown through key demonstrations and by insertion into existing national testbeds.

Table of Contents

	<u>Page</u>
1.0 Summary	1
1.1 Executive Summary	3
2.0 Introduction	5
3.0 Methods, Assumptions and Procedures	7
3.1 Architecture Description	7
3.2 Memory Stability Design/ Evaluation	19
3.3 Program Replan	20
3.4 Synergistic / Parallel Effort Investigation	21
3.5 Mathematical Benefit Analysis of Context Switching	22
3.6 Architecture Design	40
3.7 Prototype Integrated Circuit (IC) Design	41
3.8 Prototype IC Debug	51
3.9 Concept Validation Prototype (CVP)	56
3.10 CSRC Test/Demo Board (CTB)	57
3.11 VHDL Model/ Schematic Capture for Device Simulation /Verification	60
3.12 Final Integrated Circuit (IC) Development	63
3.13 Front-End Design Tools	80
3.14 Back-End Design Tools	82
3.15 Reconfigurable Computing Module (RCM)	102
3.16 RCM Board Support Software	106
3.17 RCM Demonstrations	107
4.0 Conclusions	114
5.0 References	116
Appendixes	118

Figures

<u>No</u>	<u>Title</u>	<u>Page</u>
2.1	Reconfiguration Benefits	6
3.1.1	16 Bit Data Pipe Comprised of CSLAs	8
3.1.2	Level 3 Routing Bridges Pipes	8
3.1.3	Prototype Context Switching Logic Array and Level 1 Routing	10
3.1.4	Prototype Context Switching Logic Array with Level 1 & 2 Routing	10
3.1.5	Context Switching Logic Cell Architecture	12
3.1.6	Context Switching Input/Output Cell Architecture	13
3.1.7	Private/Public Addressable Sharing Scheme	14
3.1.8	Direct / Shift Routing	16
3.1.9	Photograph of Prototype CSRC Device	17
3.1.10	Photograph of Final ("Next Generation") CSRC Device	18
3.5.1	Polyphase Filter Network Division of Symmetric Filter into Non-Symmetric Subfilters	24
3.5.2	Time Domain Beamforming	26
3.5.3	Subbeam Decomposition onto CSRC FPGA	29
3.5.4	SIMULINK CSRC Beamforming Simulation Example Output	31
3.5.5	CSRC Plume Detection Processing Block Architecture	33
3.5.6	CSRC Plume Detection Architecture	33
3.7.1	Block Diagram of I/O Buffer	43
3.7.2	Programmable Output Buffer	43
3.7.3	CSLC (Logic Cell) Floorplan	44
3.7.4	CSLC (Logic Cell) Layout	45
3.7.5	Decoders and Switchbox Layout (Level 1 Routing)	46
3.7.6	CSRC IC Layout	48
3.7.7	CSRC Prototype IC Final Layout	49
3.7.8	Photograph of Actual CSRC Prototype IC	50
3.8.1	CSRC Prototype Programming Tools	55
3.10.1	CTB Block Diagram	57
3.10.2	CTB Board	58
3.10.3	ProtoTester Screen Capture	59
3.12.1	Relative Size of Prototype (left) and Final CSRC (right) CS-Bits	66
3.12.2	Layout of CSLC in Final CSRC IC	67
3.12.3	Context Switching Memory Bit	68
3.12.4	Context Switching Memory Byte	68
3.12.5	Layout for a Byte of Dual Port Synchronous Memory	69

Figures (continued)

<u>No</u>	<u>Title</u>	<u>Page</u>
3.12.6	Context Switching Memory Decoder	71
3.12.7	Context Switching Write Decoder	72
3.12.8	Context Switching Memory 64	73
3.12.9	Layout of 256x8 Dual-Port Synchronous Block RAM	73
3.12.10	Layout of Logic Array (Including Level 2 Routing)	75
3.12.11	Layout of a Pipe (Including 8 Logic Arrays and 2 Block RAMs)	76
3.12.12	Clocktree Architecture	76
3.12.13	Context Switching Logic Array Minimized Bus Lengths	77
3.12.14	Layout of Final CSRC IC	78
3.12.15	Die Photograph of Final CSRC IC	79
3.14.1	VPR's Iterative Routing	82
3.14.2	VPR's Detailed Routing	83
3.14.3	Database Netlist Schema	84
3.14.4	Logic Equations Schema	85
3.14.5	Global Property Assessment	85
3.14.6	Timing Schema	86
3.14.7	FPGA Physical Description	87
3.14.8	Context Class	88
3.14.9	Architecture Editor Main Window	92
3.14.10	Architecture Editor Edits	92
3.14.11	User Interface Main Menu	93
3.14.12	Tool Options	94
3.14.13	Pull-Down Menus	94
3.14.14	FPGA Resource View	95
3.14.15	FPGA Post Place and Route Viewer	96
3.14.16	Net Highlighting	97
3.14.17	Critical Nets are Highlighted in Blue	97
3.14.18	Block RAMs are Fully Supported	98
3.14.19	Integration with Synplicity Anylist	98
3.15.1	RCM Block Diagram	102
3.15.2	RCM Circuit Card	104
3.16.1	GUI for Board Support Package	106
3.17.1	Wideband SAR Technology Transitions	107
3.17.2	Quadtree Algorithm	108
3.17.3	Iterative Loops of Quadtree Algorithm	109
3.17.4	Iterative Loops of Quadtree Algorithm	110
3.17.5	OpenGL View of Wideband SAR Image Formation on RCM	111
3.17.6	Utilizing PPC750 and Host Mosaicing	111
	CSRC Mapping of Quadtree Algorithm Inner Loop	113

Tables

<u>No</u>	<u>Title</u>	<u>Page</u>
3.5.1	CSRC Context Layer Sizing for Beamforming	30
3.5.2	CSRC Optical Flow Algorithm Mapping	35
3.5.3	CSRC Optical Flow Processing Requirements	36
3.5.4	CSRC Optical Flow Algorithm FPGA Resource Allocation	37
3.5.5	CSRC Optical Flow Algorithm RAM Requirements	37
3.5.6	Optical Flow Algorithm Architecture Comparison	38
3.17.1	Bit Widths and Approximation Algorithm	112
3.17.2	Determination of the Contents of each Context of the Two CSRCs	112

FOREWORD

The Defense Advanced Research Projects Agency (DARPA) is developing enabling technology to permit the effective use of advanced high performance computing technologies in Defense-critical applications. Currently DARPA is pursuing developments in reconfigurable computing components and the enabling associated software to create systems that transparently optimize and adapt their architecture to specific evolving applications and environmental constraints. The CSRC team has pursued these technologies in the recent years and has developed key components leading to technology advancements in reconfigurable computing. Sanders, for example, has developed the advanced reconfigurable computing architectures and algorithmic programming tools and has demonstrated the technology benefits using reconfigurable hardware components, optimized for computation in diverse applications including IRMW, IRST, communications (radiowaveform generation, channelizer), EW, SIGINT and sensor data routing and integration. UCLA has conducted the leading research into the exploitation of context switching reconfigurable logic devices, algorithmic designs and decomposition required by the technology and has applied the research to ATR-like applications. FPGA Technologies has developed a unique software tool set for FPGA development that is very user friendly.

Under this contract Sanders developed new models of computation based on context switching reconfigurable logic, logic devices which support the computation architectures, tools and environments which support efficient compilation, configuration optimization and design-reuse, as well as dynamic runtime configuration. Sanders shall develop and integrate the resultant hardware modules and software into advanced computational testbeds and shall demonstrate the novel computational approaches in significant applications.

The scope of this program includes exploring and developing the technology associated with context switching reconfigurable computing (CSRC) to accomplish the following: a) make the technology products available and accessible to system developers; b) demonstrate the significant enhancements in compute density afforded by such compute elements in applications of national significance, and c) to foster exploitation of the technology through insertion of developed modules and programming tools into advanced development programs and testbeds.

-

1.0 Summary

The Context Switching Reconfigurable Computing (CSRC) program focused on the development of a context switching device, design tools for these devices, design methodologies to exploit this technology, and new algorithmic techniques and associated tools to facilitate the widespread DoD and commercial application of this revolutionary device technology. The context switching device developed under the CSRC program is an FPGA that is capable of complete reconfiguration on a single clock edge, storing four configurations on chip, and allows sharing of data between "contexts" (or configurations). CSRC technology will provide the signal processing and adaptive computing community with an additional dimension of flexibility, time. It is believed that CSRC technology will result in a performance improvement of two orders of magnitude over conventional DSP and FPGA implementations for many signal processing algorithms by eliminating the need to continually move data on and off chip, instantiating application specific hardware when it is needed, and allowing for data-dependent hardware instantiation.

Development of the CSRC hardware proceeded in several stages. The first stage explored the CSRC concept, as well as guided the software tools' development effort, through the development and use of a high level behavioral model. The second stage emphasized the architect, design, fabrication, and test of a prototype logic device which supports context switching reconfigurable computing. Effort to develop new models of computation which are based on context switching reconfigurable logic and to develop software tools and models, facilitating algorithm, partitioning and mapping, as well as place and route (PPR) to foster exploitation of this revolutionary technology is ongoing.

The subsequent stage was to architect, design, fabricate, and test the final CSRC device. The device uses the prototype device as a point of departure adding features and increasing capacity. Following the development of the CSRC device technology a Reconfigurable Computing Module (RCM), which utilized the CSRC logic device, and has application to multiple computing domains including mathematical computations, image manipulation, and adaptive computing, was developed. The RCM modules and the new software will be delivered for integration as common reusable compute elements within advanced heterogeneous multi-computing testbeds and programming environments at both US government laboratories and at Sanders in an effort to widely proliferate the technological advancements afforded by this research. In addition, a design toolkit was developed to afford efficient mapping to the CSRC technology. The toolkit is user friendly, integrated with a major FPGA tool vendor, performs automatic macro generation, is capable of iterative place and routes, and provides a cross-referenced viewer.

The Context Switching Reconfigurable Computing technology has been transferred to military and commercial applications via several methods. A World Wide Web page has been maintained to provide access to technical papers, trade studies and benchmark results, designs, tools, documentation and design notes. Papers have been, and will continue to be, published at key technical conferences. Technology developed under the

CSRC program will be integrated into testbeds residing in the United States National Laboratories to make the technology available for use by other researchers. Finally, Sanders has been pursuing a commercialization strategy to facilitate widespread proliferation of CSRC technology.

1.1 Executive Summary

A team led by Sanders, a Lockheed Martin Company, was awarded a 36 month contract to develop a context switching reconfigurable computer (CSRC) device. The CSRC is able to process certain classes of signal processing algorithms with up to 100 times greater performance over today's microprocessor-based systems. Sanders teammates include Xilinx, Inc., of San Jose, California, (during the early years of the program), and the University of California at Los Angeles, and FPGA Technologies of San Jose, California. The project was administered through the U.S. Air Force Materiel Command, Rome Laboratory, Rome, NY, work was performed at Sanders facilities in Nashua, N.H., and was completed with a no cost extension in July of 2000.

Under the contract, the team demonstrated that CSRC devices can improve compute density by a factor of up to one hundred by means of a "hardware context switching" technique. Unlike a traditional microprocessor which relies on fixed hardware logic; controlled by instruction decode logic, the CSRC has a number of reconfigurable "logic" configurations or contexts stored on the device itself. Each configuration stores the equivalent of a mini-program that can be executed in hardware logic. By rapidly switching through the configurations, a technique called context switching, the device appears like it is processing the mini-programs simultaneously. In this manner the CSRC is like a multi-tasking computer which appears to be running programs simultaneously, but with a unique twist. The device allows data stored in registers to be shared between the contexts. In this manner results produced by one context are automatically available to another context without any data movement. The combination of implementing the mini-programs in hardware and minimization of data movement account for the performance improvements that can be achieved.

The design of the device allows the hardware in each context to be re-programmed or re-configured to meet the processing needs of the particular program to be processed. This means that you can approach the performance of a custom circuit while having the flexibility to support a wide range of algorithms. In addition, the device is able to reconfigure a context while it is processing another context. This dynamic reconfiguration allows the device to look like it virtually has an infinite number of contexts. Analysis has shown how context switching hardware can improve performance in several computing domains, including mathematical computations, image manipulation and adaptive computing.

Sanders' led the CSRC device architecture development; trade-off studies; algorithm mapping and system programming tool development; and design of prototype modules. Xilinx was responsible for developing the CSRC devices, based on its field programmable gate array technology, and the device-level programming tools. UCLA provided expertise in the development of reconfigurable architectures and their customization for ATR applications.

Initially the emphasis for the development was on determining the features and the computational model of the new context-switching device. Information was solicited from numerous external sources, as well as many internal Sanders and Lockheed Martin

programs, in an attempt to determine a feature set that best suits a wide range of problems. When a feature set was finalized a design review of the CSRC device was hosted by Xilinx.

The review formalized the results of the feature solicitation and tradeoff studies, the investigation of potential computational models for the CSRC device, as well as a broad market survey. Based upon these findings, a CSRC device architecture was formalized. Although Xilinx had initially committed to developing a multi-layer context-switching device that would be used by the CSRC program, they indicated that they felt that incorporating reconfigurable computing features into the next generation Xilinx devices was a better technical solution than creating a unique multi-context device. Although the new device that Xilinx had proposed would have allowed new areas of reconfigurable computing to be explored, the concept of moving the instructions to the data could no longer be accommodated.

At this point, under customer direction, Sanders accepted the responsibility of developing the multi-layer context switching device. Sanders selected a fine grain architecture based upon its extensive past experience with FPGA technology and due to the fact that lent itself more readily to the incorporation of features sought by potential users as compiled in the feature "wish list". As the program was re-focused, the customer emphasis was placed on a two-phase development. The first phase would develop a small prototype device as a proof of concept and the second phase would produce a larger chip, the development tools and a module that would contain CSRC devices so that the technology could be demonstrated.

Both the proof of concept and final device have been designed, fabricated and tested with a first pass success. A reconfigurable computing module (RCM) was designed fabricated and tested successfully. The RCM fits into a standard computer PCI slot and contains two CSRC devices. The RCM has been integrated into a PC environment so that host programs can use the RCM to demonstrate CSRC technology. A sophisticated suite of development tools has been successfully built so that designers may describe circuits and map them seamlessly onto the multi-layered contexts of the CSRC device. The new DARPA TTO sponsored DRACs project will employ the CSRC device, the RCM, and the CSRC Toolkit to further enhance dynamic reconfigurable technology by affording the designer a design environment that facilitates the ease of developing runtime reconfigurable systems.

2.0 Introduction

History has seen the methodologies of computing evolve from *fixed* hardware and *fixed* software (ENIAC), to *fixed* hardware and *reconfigurable* software (microprocessors), to *reconfigurable* hardware and *reconfigurable* software (FPGAs). FPGAs have traditionally been utilized in applications that demand the performance of application specific integrated circuits (ASICs) while maintaining the flexibility and rapid design cycle afforded by the use of digital signal processors (DSPs). Although FPGAs are not ideally suited for either requirement, they do offer an excellent compromise. In the recent past, many research efforts have examined the possibility of performance enhancement due to run-time reconfiguration. However, the best of today's commercially available technology requires milliseconds to reconfigure. This reconfiguration time, although acceptable for some applications, such as the *SPEAKEasy* reconfigurable "softradio" developed by Sanders, is an unacceptable delay for most real-time systems. Although partial reconfiguration can reduce the required reconfiguration time, this is believed to be an alternative approach to dynamic reconfiguration. Being able to *completely* reconfigure an FPGA at a rate that far exceeds the necessary persistence of a hardware function, while being able to share data between configuration instantiations is believed to be tomorrow's reconfigurable computing computational model. This model of computation shall be referred to as *context switching reconfigurable computing* and is a natural extension of today's methodology. Arguably, context switching is not unlike the very first mode of computation. In essence, clock-cycle dynamic reconfiguration can be viewed as *fixed* hardware and *fixed* software since the available hardware can be thought of as being virtually infinite – *virtual hardware*.

The context switching reconfigurable computing (CSRC) technology being developed by Sanders extends commercially available field programmable gate array (FPGA) devices to include high speed changes between a number of programmed functions without the need for additional FPGAs. Each configuration, referred to as a *context*, in a CSRC FPGA has the functionality similar to that of many commercially available FPGAs. The context switching can occur at significantly higher speeds than the rate at which current FPGA technology can reconfigure. In addition, unlike commercial FPGAs, where reprogramming destroys any resident data, the CSRC FPGA affords the capability of data sharing between contexts.

The concept of virtual hardware is an obvious benefit of dynamic reconfiguration. If configurations can be swapped in and out of an FPGA upon demand at a real-time system rate, only the necessary hardware need be instantiated at any given time. In this manner, a virtually infinite algorithm cache or an infinite coprocessor can be conceived. In other words, a high level system scheduler can instantiate hardware as needed. In this manner, a reduction in size, weight, and power can be achieved. Additionally, given the CSRC FPGA, if the processing requirements specify a sequential application of algorithms, the context layers can be set up to share data such that the output of one algorithm is immediately available as the input to the next algorithm upon a context switch. This is not possible with contemporary FPGAs.

A natural extension of the algorithm cache mode of computation is the concept of mission phase reprogrammability. As seen in Figure 2.1, an entire mission can be mapped to a CSRC device. In this case, different contexts can house different algorithmic phases of a mission without requiring that an algorithm be confined to a single context, depicted as layers in Figure 2.1.

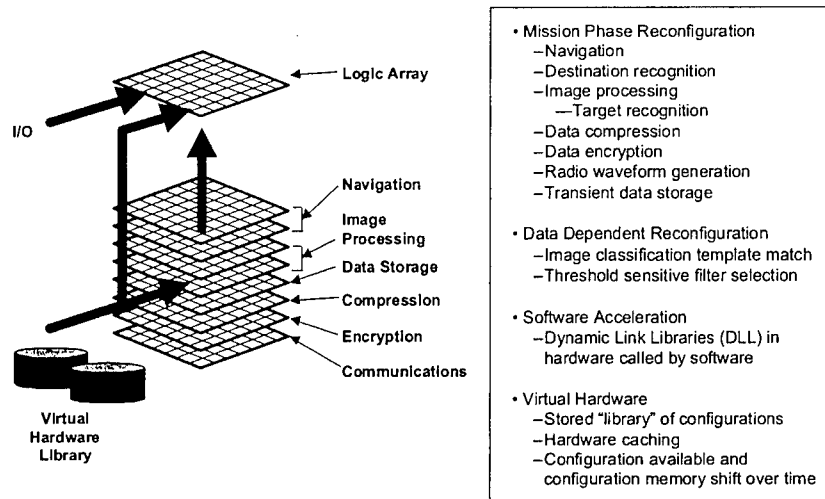


Figure 2.1: Reconfiguration Benefits

Although Figure 2.1 identifies the obvious modes of computation for gaining a performance enhancement, it is believed that the true potential of context switching requires a paradigm shift in algorithm implementation. The capabilities of the CSRC architecture, which extend dynamic reconfiguration to context switching, have the potential to provide improved implementations of signal processing algorithms over those currently available through commercial FPGAs. The inherent ability of CSRC to quickly perform different tasks and share results among different configurations allows one to approach algorithms from a different perspective, enabling mathematical implementations previously inconceivable without context switching.

Although the past few years have seen much interest in context switching reconfigurable computing, it is believed that this document describes the first design and implementation of such a device. Up until now, all of the substantiated work on this model of computation has been theoretical. The emphasis of this document is to reveal the architecture of the context switchable FPGA being developed by Sanders. It is hoped that this document will spark new ideas and facilitate algorithmic research that is targeted towards a specific and real architecture. With this achieved, as the world's first context switchable silicon becomes available, members of the adaptive computing systems (ACS) community will be capable of taking full advantage of this new technology.

3.0 Methods, Assumptions and Procedures

The Context Switching Reconfigurable Computing (CSRC) contract was awarded to Sanders. Sanders was authorized to proceed on the CSRC base program and on the Memory Stability Design / Evaluation option. Teammates included Xilinx, Inc., of San Jose, California, the University of California at Los Angeles, and FPGA Technologies Inc, of San Jose California. The project was administered through the U.S. Air Force Materiel Command, AFRL, Rome, NY. This section will depict the significant details of the effort that has ensued since award of contract.

3.1 Architecture Description

Experience has shown that FPGAs afford the greatest performance benefit when they are used to implement algorithms with deep pipelines. However, pure dataflow algorithms are rare. In fact, generating pipeline control signals, implementing state machines, and interfacing with external RAM or other integrated circuits, are critical, although not typically areas of performance enhancement, to an FPGA's successful system integration. With this in mind, the CSRC device was designed to be a 4 bit DSP dataflow engine that is simultaneously capable of efficiently implementing glue logic. However, since FPGA performance enhancements are oftentimes achieved by implementing the minimum required bitwidth, the CSRC device was developed to allow users to implement scalable pipelines such that the wordwidth can be of any size.

3.1.1 Data Pipes

The CSRC device is arranged into 16-bit wide data pipes. Each pipe is formed by a plurality of context switching logic arrays (CSLAs) as seen in Figure 3.1.1. A single CSLA is capable of processing two 16-bit words and outputting a 16-bit result. The result of a CSLA is available as an input to the two adjacent CSLAs in the pipe. Hence, a pipe can naturally be used as a data path. Information can easily flow from one end of the pipe to the other. It is important to point out that in this device data can non-preferentially flow in both directions. This feature has great utility when sharing data among different contexts. For example, one context could process data from left to right, storing it's final result in the right-most set of registers. Note that it is quite possible that the final result of a single context is actually an intermediate result of the entire algorithm. Given this situation, an incoming context can pick up where its predecessor context left off by acquiring the intermediate data deposited on the rightmost portion of the pipeline and processing it in a pipeline that flows from right to left. From this simple example, it can be seen that a data path that does not favor data flow in either direction, is more efficient for context switching hardware because it alleviates the need to reroute data from its physical origin in one context to its physical input in the subsequent context.

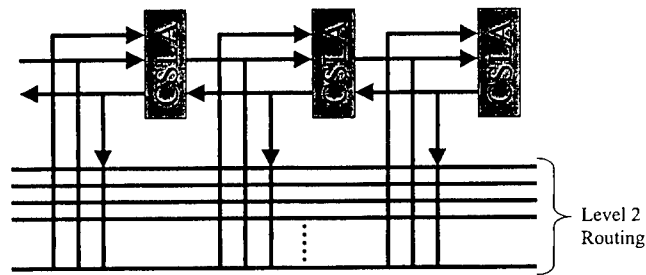


Figure 3.1.1: 16 Bit Data Pipe Comprised of CSLAs

Level 2 routing can be found alongside the pipe and consists of 16-bit buses. See Figure 3.1.1. These busses are not segmented and run the entire width of the CSRC device. This type of bussing scheme implies that a signal driven onto level 2 routing is available to any CSLA in the pipe. Additionally, this approach affords the possibility of faster and less complicated programming tools than segmented approaches because the timing is more deterministic. Each CSLA has two 16-bit inputs, each of which is capable of tapping into any of the Level 2 routing busses. Similarly, the CSLA's 16-bit output can drive any of the Level 2 routing busses. Note that Level 2 routing can be utilized as a bus architecture, can be broken down and utilized by individual bits, or can be employed as any combination of these.

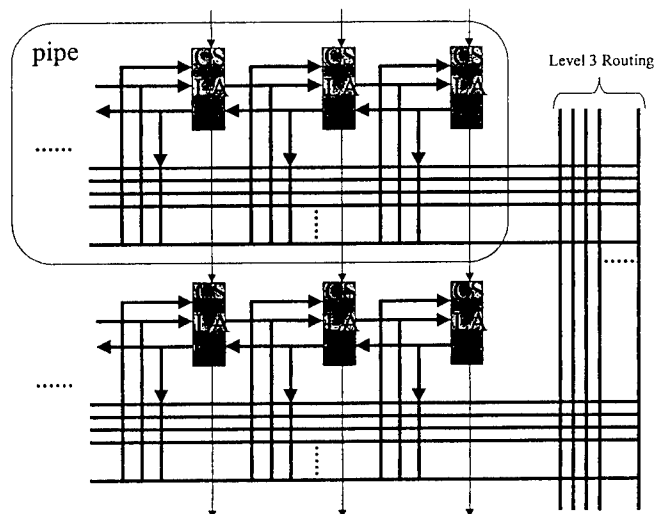


Figure 3.1.2: Level 3 Routing Bridges Pipes

The CSRC device is formed by stacking up pipes one on top of the other. Corresponding CSLAs on adjacent pipes have dedicated wiring that allows them to pass along their carry bit. This feature allows two adjacent pipes to be bundled together and be used as a single 32-bit wide data path. In actuality, physical 16-bit pipes can be broken down into smaller logical pipes. Although hardware is optimized to break pipes into nibbles, pipes can be n-bits wide.

As seen in Figure 3.1.2, information driven onto a given pipe's Level 2 routing can be connected to Level 3 routing which in turn makes the data available to any Level 2

routing on the chip. Similar to the Level 2 routing structure, the Level 3 routing is not segmented and spans the device. Note that conceptually the Level 2 and Level 3 routing are perpendicular to each other.

I/O pins on the device are connected to Level 2 and Level 3 routing. All pins physically located on the top and bottom edges of the device connect to Level 3 routing. Pins on the left and right edges can connect to either Level 2 routing or directly into the dedicated routing that normally connects adjacent CSLAs.

3.1.2 Context Switching Logic Array

A single CSLA is primarily composed of 16 context switching logic cells (CSLCs) and Level 1 routing to interconnect them. Figures 3.1.3 and 3.1.4 depict a CSLA and the CSLA as it attaches to the Level 2 routing, respectively. Note that the routing structure depicted applies to the prototype IC. The final IC routing architecture is slightly more flexible but utilizes similar structure to that employed in the prototype IC. The CSLCs are numbered 0 through 15 and their carry-in and carry-out chains are hardwired appropriately so they can function as a single cohesive unit. Level 1 routing consists of three 16-bit busses. Two of these 16-bit busses are inputs from the Level 2 routing. The third 16-bit bus is hardwired to the outputs of the CSLCs. Level 1 routing was designed with two modes of operation in mind.

3.1.3 Routing Modes of Operation

As previously mentioned, it is believed that the most beneficial FPGA is capable of exploiting its inherent DSP strengths while simultaneously being capable of implementing the often required glue logic. Hence, the CSRC FPGA has been designed with two modes of operation in mind: (1) Deep pipeline mathematical operations that can be of arbitrary bitwidth & (2) Random logic implementations that encompass control, state machines, and interfacing with external RAM or other integrated circuits. As a direct result, the CSRC FPGA exhibits two types, or modes, of routing.

3.1.3.1 Bus Routing

The first operational mode of routing is bus routing. The design goal was to provide users with the ability to route entire 16-bit words in and out of CSLAs while maintaining bitwidth order (i.e. the most significant bit (MSB) in the MSB position and the least significant bit (LSB) in the LSB position.)

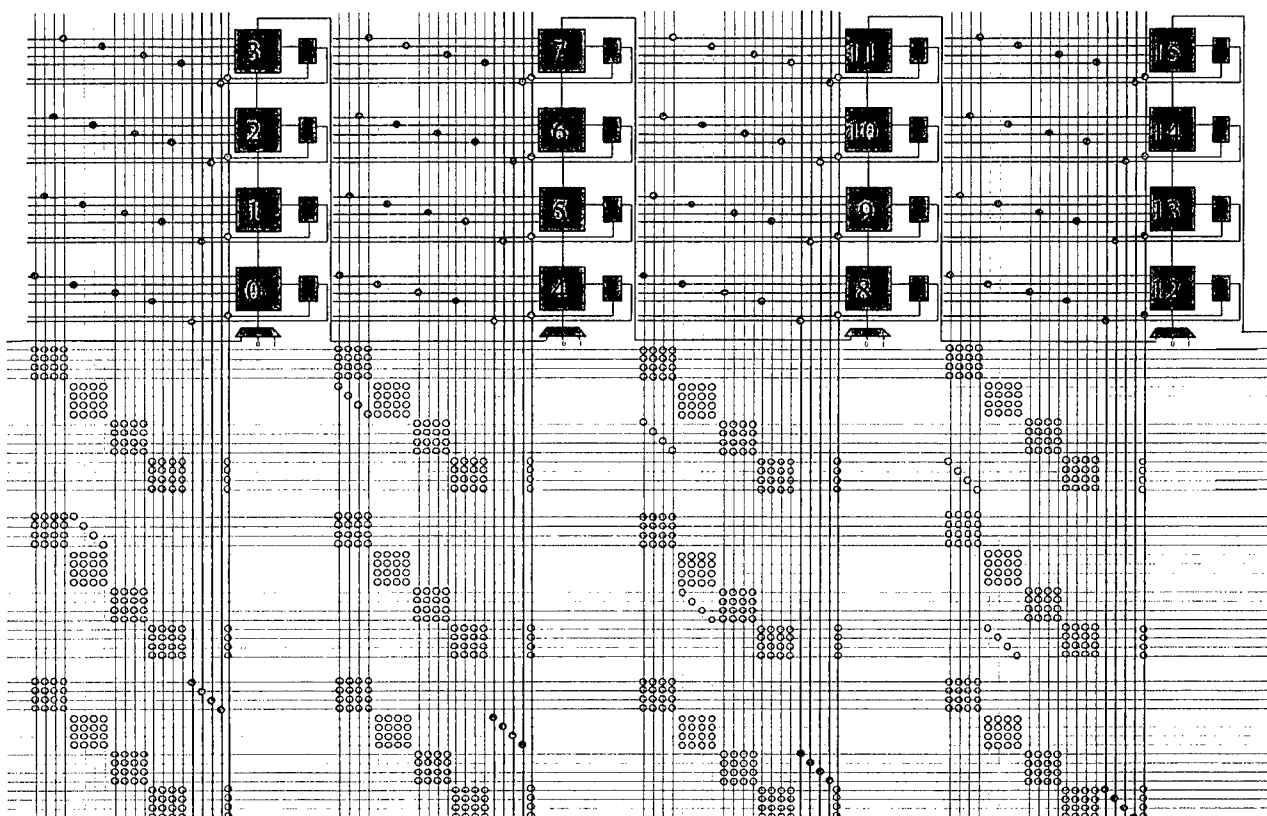


Figure 3.1.3: Prototype Context Switching Logic Array & Level 1 Routing

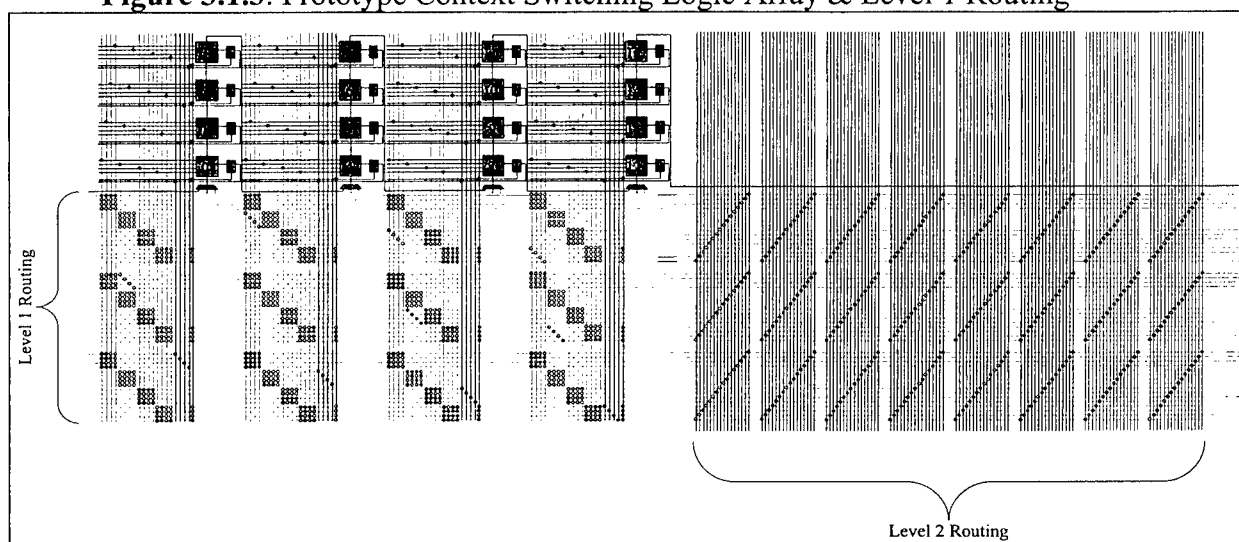


Figure 3.1.4: Prototype Context Switching Logic Array with Level 1 & Level 2 Routing

Given that the four data inputs to the CSLC are labeled as A, B, C, and D, enough programmable connections are contained in the Level 1 switching matrices to ensure that one of the input buses can be routed into the A inputs of all of the 16 CSLCs. The least significant bit of the bus feeds the A-input of the least significant CSLC and so on. In

essence, this bus can be considered the A-input (16-bits wide) for the entire CSLA under this bus routing mode of operation. Note that the second 16-bit bus can be used to feed the B inputs of the CSLCs within a CSLA in a similar fashion. The final bus connection is hardwired to the 16-bit output of the CSLA and attaches to the Level 2 routing. Note that this output is also a direct connect between neighboring CSLAs. As previously described, this non-directional direct connect allows for fast routing between CSLAs within a pipe by alleviating the need for Level 2 routing if the output of a pipe stage is feeding a neighboring CSLA.

3.1.3.2 Bitwise Routing

The second mode of operation is bitwise routing. No matter how data processing intensive a design might be there is almost always a need for control logic whether it is simple glue logic or more complex state machines. For this reason the bitwise routing mode of operation is necessary. The basic premise is that the output of any given CSLC within a CSLA should have at least one possible path to connect to at least one input of all other CSLCs within the same CSLA. A simple pattern of programmable connections was developed to enable this feature. All the A-inputs of all the CSLCs in a CSLA can tap into the four least significant bits of all three Level 1 routing busses (this includes the output bus to provide a means of local feedback without having to waste Level 2 routing resources). Similarly the B-inputs and the C-inputs tap into the next 4 bit bundles within each level 1 routing bus, and finally the D-inputs tap into the four most significant bits on every bus. As a result, the four least significant CSLCs, which drive the corresponding four-least significant bits of the output bus, are capable of driving any A-input on any CSLC within the same CSLA. For this reason these four CSLCs are known as "A-drivers" under the bitwise routing mode of operation. Similarly, B-drivers refers to CSLCs 4 through 7, C-drivers to CSLCs 8 through 11, and D-drivers to CSLCs 12 through 15. Furthermore, since connections between Level 1 and Level 2 and connections between Level 2 and Level 3 maintain proper bit order (LSBs to LSBs and MSBs to MSBs) any A-driver can drive the A-input of any CSLC anywhere in the chip. For these same reasons, the same functionality applies to the B, C, and D-drivers.

In addition to the four main inputs (A, B, C, & D), each CSLC has a clock enable / tri-state control line. Both of these control lines tap into the four most significant bits of the three Level 1 busses, hence, they are controlled by D-drivers. As seen in Figure 3.1.5, the clock enable / tri-state control line is a single control line to the CSLC. For this reason, the user can choose to use this control line to control either the clock enable or the tri-state buffer. Note that in the final CSRC IC, the tri-state functionality has been removed leaving only the enable control signal.

3.1.4 CSLC

The CSLC is the heart of computation for the CSRC device. As seen in Figure 3.1.5, the CSLC is composed of carry logic, a four input lookup table (CSLUT), a context switching flip-flop (CSFF) and a tri-state buffer. The carry logic unit is capable of generating carry bits for either additions or subtractions. The carry logic chain is connected by dedicated connections. The chain can be connected, disconnected, or fed a logic zero or logic one every four bits. In this manner, the bus routing mode can be

utilized to generate a pipeline granularity of four bits. However, in reality, the buswidths can be of an arbitrary bitwidth, n . Note that bitwidths with a modulo 4 = m , where m is greater than zero, will disallow m CSLCs from supporting a mathematical pipe that requires the starting of a carry chain.

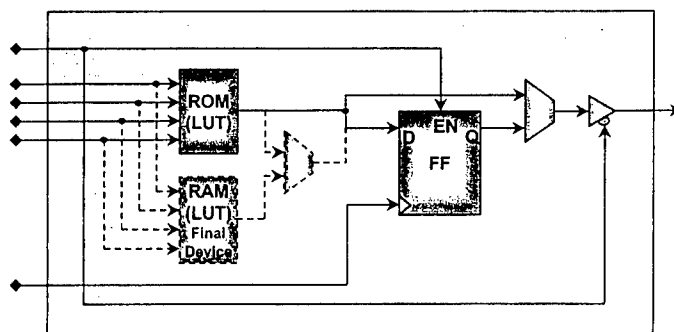


Figure 3.1.5: Context Switching Logic Cell Architecture

The outputs of the carry logic feed the CSLUT which consists of 16 context switching configuration bits (CSBits) that are multiplexed together. The 4 inputs serve as the select lines therefore implementing a programmable function. Note that the contents of each of the CSLUTs are unique in each context and specified in the configuration bitstream.

The CSBits implement context switching itself. Each CSBit holds a single programming bit for every context. However, only the active context's value drives whatever logic the CSBit is controlling. A detailed description of the context switching functionality afforded by these CSBits is described in section 3.1.6

Unlike some commercial FPGAs, the lookup table can not serve as a memory element because the CSLUT is composed of CSBits, not SRAM. Instead a separate context switching RAM (CSRam) provides memory storage facilities. The CSRam, which is only available in the final CSRC device, implements the *global sharing scheme*. This data sharing scheme is similar to traditional blackboard data sharing. Any data written to a CSRam memory is available to all the CSRam elements that are physically collocated among different contexts. The data value last written into the active CSRam, before deactivation of the current context, will be seen by all other collocated CSRams upon the activation of their respective contexts. In fact, one can envision writing to a CSRam in one context and having its contents be used as a LUT in another context. Additionally, it is the CSRam that will allow for large amounts of data passing between contexts to facilitate modes of computation such as moving the algorithm through the data. This mode of computation is advantageous; due to the fact that the on/off chip accesses are minimized by loading the data on chip, and keeping it on chip, until the entire algorithm has been run on the data.

Both the CSRam and the CSLUT coexist in the final CSRC device and their outputs are multiplexed together. The select line of this MUX is yet another control line to the CSLC and it is connected to Level 1 routing in the same fashion as the clock enable / tri-state control (driven by D-drivers). The output of this MUX can then be registered or passed

directly out of the CSLC as seen in Figure 3.1.5. Note that if the data is to be registered, it will be done in the context-switching flip-flop (CSFF). During regular operation within a single context, the CSFF appears to the users as a normal D-flip-flop (DFF). The DFF connects to the global clock and it is controlled by the clock enable input to the CSLC.

3.1.5 CSIO

The context-switching input/output cell is used to facilitate on/off chip data accesses. As can be seen in Figure 3.1.6, the CSIO cell is bi-directional, can provide latched or direct outputs, and has a programmable pull-up resistor on the output. In addition, the CSIO cell can tri-state its output. Since on/off chip access time is oftentimes a limiting factor of FPGAs, a programmable drive strength capability has been included to insure maximum performance. Finally, the flip-flop in the CSIO cell utilizes a different sharing scheme than the flip-flop in the CSLC. Note that since it is believed that sharing data between contexts within a CSIO cell is unlikely to be a key feature, the global sharing scheme is implemented for the CSIO cell DFFs rather than the more complex sharing scheme that is implemented in the CSLC's DFF.

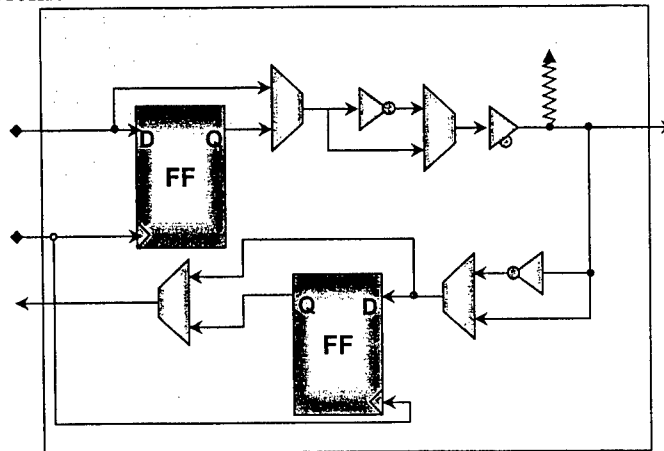


Figure 3.1.6: Context Switching Input/Output Cell Architecture

3.1.6 Data Sharing / Context Switching

Research indicates that the major benefits of context switching are afforded by sharing data between contexts and being able to switch between contexts very rapidly. For this reason, a great emphasis has been placed on the development of a device that meets both of these needs. The two sharing schemes that have been designed and implemented are Global Sharing and Private/Public Addressable Sharing (P/PASS). The global sharing scheme is used in the CSIO DFF and in the CSRam while P/PASS is used in the CSLCs by the CSFF. Global Sharing, as previously described, is simply a common memory element between all contexts. Hence, all contexts view these same memory elements and when any context writes to the memory element, the change is seen by all contexts upon their respective activation.

The data sharing scheme used by the CSFF, P/PASS, truly exposes the novelty of the CSFF and is depicted in Figure 3.1.7. With this type of sharing, each CSFF within each context supported in hardware has a corresponding register. These registers are known as

private registers since they belong to a particular context and can only be accessed by a specific DFF within the context. Additionally, there is a single active register per CSFF. The active register is what the user actually utilizes during uninterrupted context execution. Upon switching contexts, the outgoing (active) context saves its intermediate values to its private registers. This feature enables many of the capabilities that would be needed to develop secure kernels by isolating intermediate data. Additionally, a context can *choose* to write its values to a *public* register (on a Logic Cell by Logic Cell basis) which can be addressed by any and all of the contexts. In this manner, the sharing of data between DFFs within contexts is enabled. The number of public registers available in a P/PASS implementation is independent of the number of contexts supported directly by hardware. Hence, public registers must be addressed when used. Note that the CSRC device affords two public registers. Upon activation, a context can choose to restore its previous state by reading from the private register or it can opt to load a state from either public register (on a Logic Cell by Logic Cell basis).

P/PASS provides a means to keep secure data isolated while at the same time allowing data to be shared (if so desired) using public registers. This architecture scales to implementations with more contexts than hardware supports, allows sharing data between contexts that do not necessarily follow one another in time, and provides a clean and solid foundation to add features such as interrupt handling and hardware recursion.

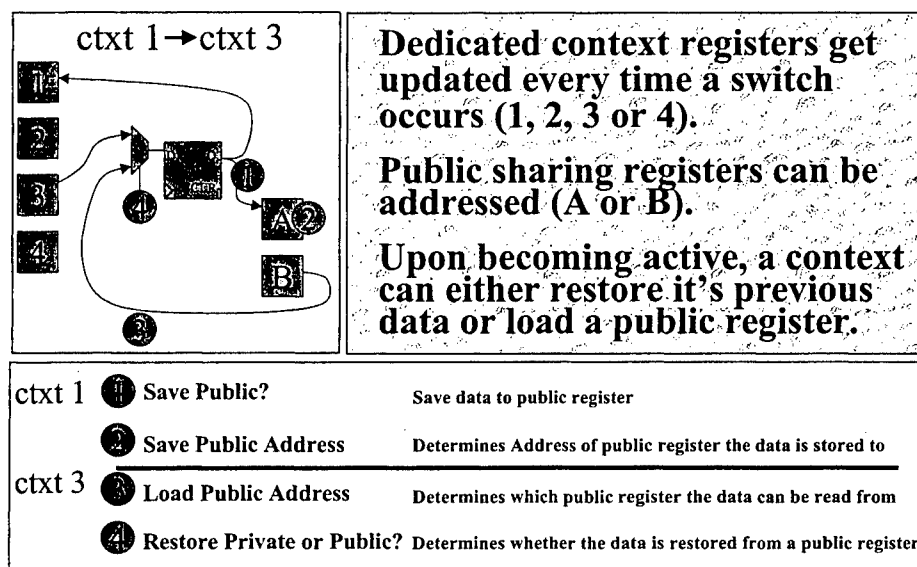


Figure 3.1.7: Private/Public Addressable Sharing Scheme

Since some modes of computation, such as the virtual coprocessor, require rapid reconfiguration, the CSRC device was designed to be capable of switching contexts on a single clock cycle. A key point to be made is that this single cycle context switching not only includes completely reconfiguring the CSRC device but completely executing all of the data sharing schemes. In fact, the active context can be swapped so rapidly that a context can be processing data on one clock edge, switch to a new configuration (including data sharing) and be processing data in the new configuration on the very next

clock edge. SPICE simulations indicate that it is possible to switch contexts in fewer than 5 nanoseconds. A caveat to this rapid context switching is that time will be required to distribute the "switch to" lines throughout the chip. These lines indicate which context the device is supposed to switch to upon receiving the "switch" signal. However, given that the "switch to" lines are stable, the context switch can take place as described above. Note that this delay in switching is merely a latency and can therefore be factored into the logic that initiates a switch. Since the switch can be initiated by the active context or via external stimulus this latency is easily accounted for.

3.1.7 Block RAM

One of the new features added to the prototype CSRC device is the block RAM. The final CSRC device has two 256x8 dual port synchronous RAM blocks per pipe. Since there are eight pipes (each with 8 CSLAs in each pipe), there are 16 block RAMs in total. Traditionally, commercial FPGA vendors tend to employ either block RAM (Altera) or distributed RAM (Xilinx). However, Sanders experience has shown that both types of RAM are valuable to computation. For this reason, the CSRC device employs both types of RAM. In fact, the CSRC device architecture emerged *prior* to commercial devices, such as the Xilinx Virtex family, now utilize both distributed and block RAMs.

3.1.8 High Speed Direct Connect Routing

Subsequent to developing the prototype CSRC FPGA architecture it was determined that a means for implementing constant coefficient multiplies more efficiently. Figure 3.1.8 depicts the additional routing developed for the final CSRC IC. As can be seen, the result of a logic cell can be forwarded to its counterparts on adjacent logic array or even fed back to itself. Optionally; the forwarded result of a logic array can be shifted down by 0-7 bits before forwarding. Since the relative routing delay is directly proportional to the level of routing (I, II, or III), it is advantageous to keep routes in lower level routes. Without this "fast routing", communication between CSLAs requires the use of Level II routing. However, if data is routed between adjacent CSLAs, this direct routing can be used which alleviates the need to get onto Level II routing. Hence, routing delays go down and performance goes up.

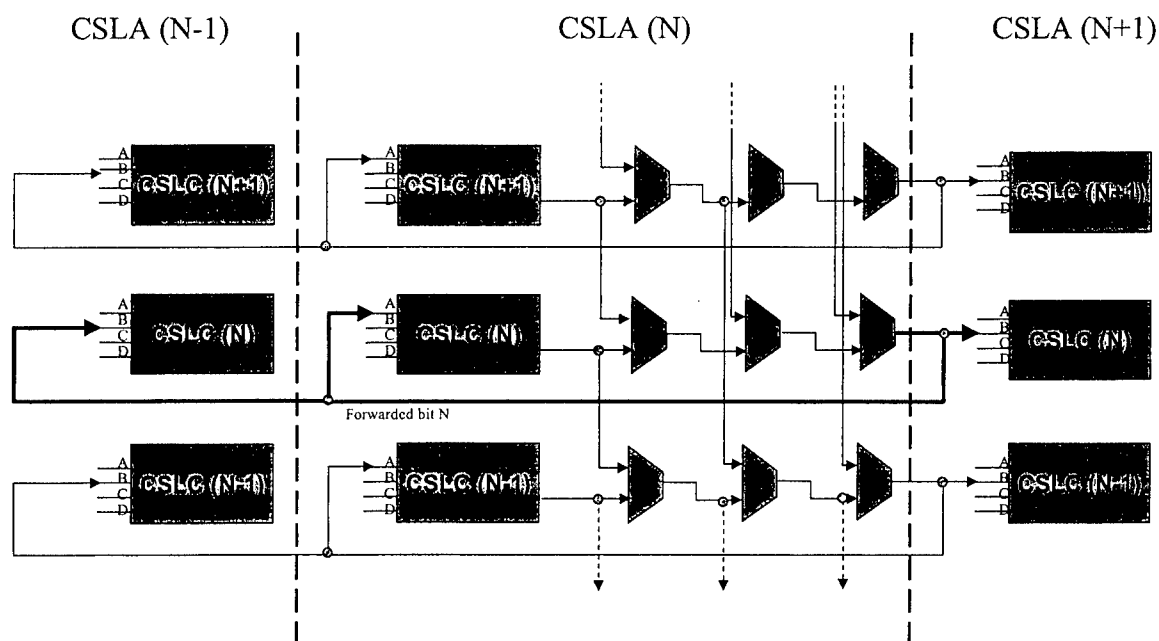


Figure 3.1.8: Direct / Shift Routing

3.1.9 Programming

The bitstreams for the CSRC FPGAs are downloaded serially (8 bit parallel for the final CSRC device). The user is required to specify which context is about to be downloaded and then supply a clock and data. By repeating this process four times, the user can configure all four on-chip configurations. Note that the configuration being downloaded can not be active during configuration download. However, inactive contexts can be downloaded while another context is active and running. Additionally, a bitstream may be downloaded by the active context. In this manner, one can envision the possibility of passing compressed or encrypted bitstreams into the active context so that it may download an inactive context after uncompressing or decrypting the bitstream.

The device will power up, prior to downloading bitstream(s), in a known state possessed by all four configurations. This provides the user with the ability to determine if the device is operational prior to use. This "known state", is both benign and affords built-in self-test (BIST). A random number generator passes data through *all* of the CSLCs and compares the results at the output, indicating pass or fail on an output pin. This pin can be monitored to verify that the device is functioning properly.

This picture is the actual die photograph of the CSRC prototype FPGA design. The design is a full custom, 0.35u design developed entirely at Sanders. The 0.5 million transistor design was fabricated at National Semiconductor

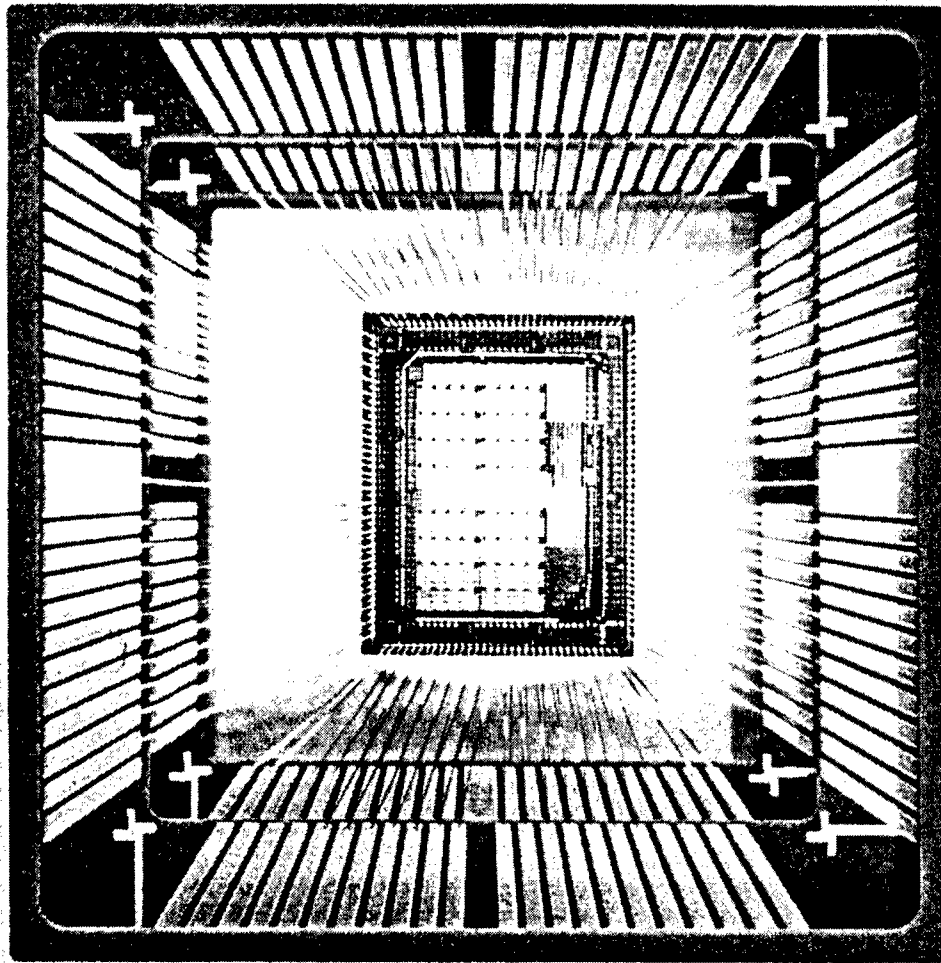


Figure 3.1.9: Photograph of Prototype CSRC Device

The full function *Next Generation* CSRC Architecture device is a 0.35u (4 layer metal) full custom design that has more than 8.5 million transistors.

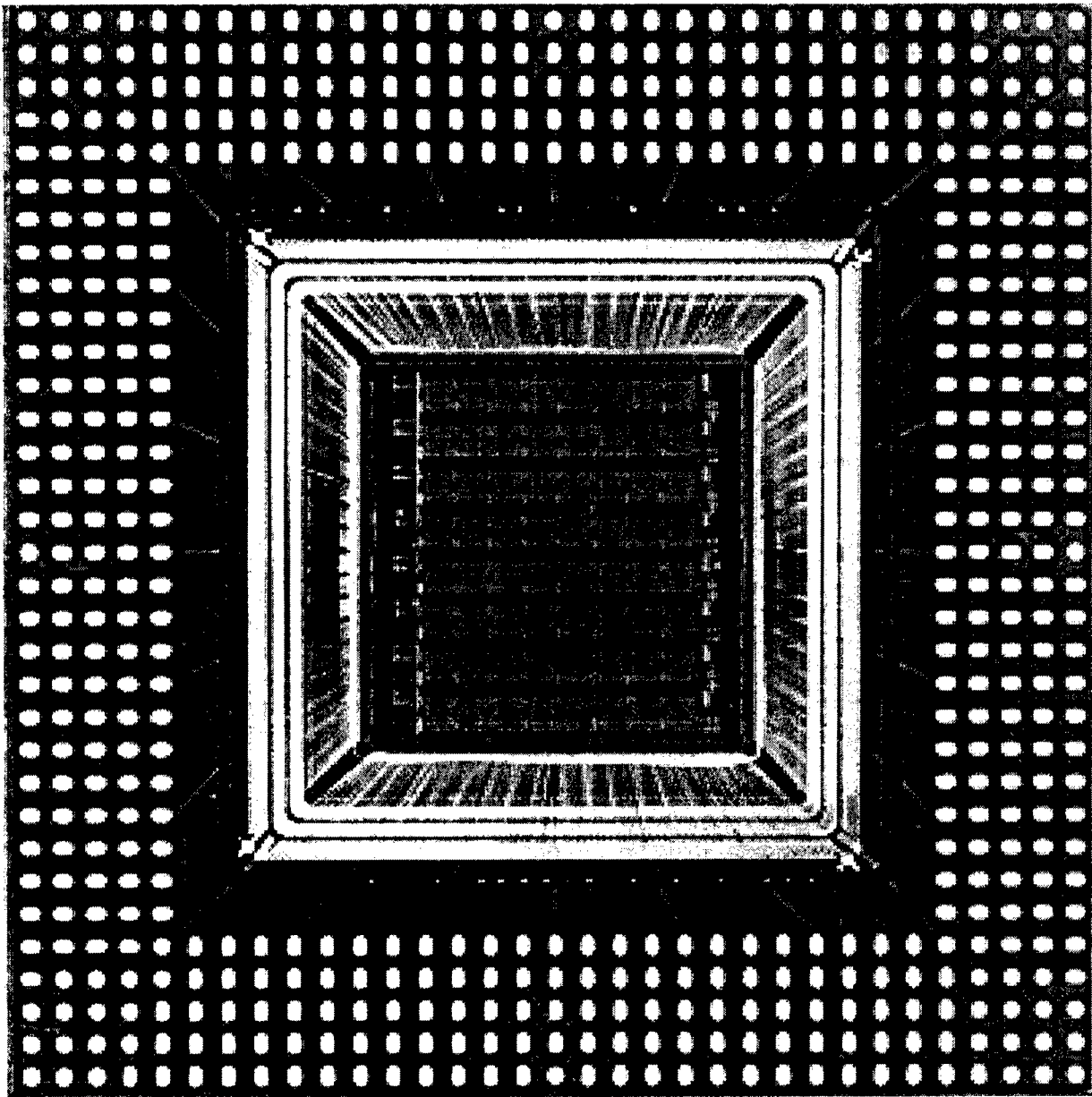


Figure 3.1.10: Photograph of Final (“Next Generation”) CSRC Device

3.1.10 CSRC Control Block

The CSRC Control Block supports both external and internal programming as well as internal and external switching requests. Programming requests are denied if an attempt is made to program an active context because active contexts can not be programmed. A switching request is denied if an attempt is made to switch to a context that is either currently being programmed, not programmed, or is being requested for programming. An external switch will always take precedence over a simultaneous internal switch request to ensure that the user does not lose control of the device. Once the programming request of a context has been submitted and accepted, the control block will look for a preamble pattern in the bitstream to sync to. From that point on, the bitstream will be

passed on the context being programmed. A counter will detect the end of the bitstream and complete the programming sequence.

3.2 Memory Stability Design / Evaluation

In an attempt to better understand the level of radiation hardening required for the user community, Sanders worked with JPL/NASA. With this understanding, Sanders compared the current Xilinx process with current Rad Hard processes to develop the CSRC Rad Hard approach. Note that Sanders baseline for the CSRC device was based upon the Xilinx XC4000 family. See Section 3.3 for details on this technical departure. However, prior to this technical change of direction, Sanders examined the Lockheed Martin Federal Systems Group (Manassas, Virginia) Rad Hard process to determine the feasibility of mapping an existing Xilinx 4K series FPGA to their foundry. The current Xilinx design requires a 0.6 μM metal width and a 0.6 μM spacing and the best existing Lockheed Martin Federal Systems Group process requires 0.8 μM for the metal width and 0.8 μM for the spacing. This will make it unlikely that a direct mapping of the existing Xilinx process can be mapped to the Lockheed Martin Federal Systems Group process. The differences between the two processes are great enough to preclude some type of scaling of the Xilinx process to the Lockheed Martin Federal Systems Group process. Processes from other foundries including Honeywell were evaluated for compatibility with the current Xilinx process.

The CSRC program met with Honeywell to discuss creating a Radiationed Hardened version of the Reconfigurable Computing device that will be used on the CSRC program. Honeywell has a process today that may be a candidate process for the existing Xilinx designs. This task was limited to a study due to funding constraints.

3.3 Program Replan

The revised plan stated that Sanders would develop a multi-context FPGA based on the logic cell design and interconnect design work performed by the University of California at Berkeley. A prototype device was proposed to be fabricated using Mosis to validate the logic cell and the first level of interconnect. Once validated, the entire context switching device would be designed and fabricated using a commercial foundry.

3.4 Synergistic/Parallel Effort Investigation

Sanders has kept a watchful eye for any sources of synergy with other DARPA contracts. The efforts that were evaluated follow:

1. *Cached Virtual Hardware for Configurable Computing* –CarnegieMellon University
2. *Programming Approaches for Run-Time Reconfigurable Systems* - Brigham Young University
3. *System Level Applications of Adaptive Computing* – USC Information Science Institute
4. *Algorithm Analysis and Mapping* –Sanders, A Lockheed Martin Company
5. *Development of a Nonvolatile, Reconfigurable, Radiation Hardened, Field Programmable Gate Array (A Novel Field Programmable Gate Array for Space Applications)* – Mission Research Corporation / DSWA via Phillips Laboratory

3.5 Mathematical Benefit Analysis of Context Switching

3.5.1 Introduction

Context switching reconfigurable computing (CSRC) technology extends commercially available field programmable gate array (FPGA) devices to include high speed changes between a number of programmed functions without the need for additional FPGAs. Each context layer in a CSRC FPGA has the functionality of an FPGA and the data resident on-chip can be shared between the context layers. The context switching can occur at much higher speeds than it would currently take to reprogram an FPGA. In addition, reprogramming a commercial FPGA destroys any resident data. These added capabilities of the CSRC architecture have the potential to provide improved implementations of signal processing algorithms over those currently available through commercial FPGAs.

In this math study, a number of candidate algorithms were assessed according to potential benefits from a CSRC implementation. Two of these candidate algorithms, time domain beamforming and optical flow, were analyzed in more detail to size the applications to available CSRC components and highlight the benefits of the CSRC technology over commercial FPGA devices. Note that at the time of this study, the architecture of the CSRC was still evolving. For sizing purposes, estimates were made on the projected CSRC capacity. This section should be read with the understanding that the emphasis is truly how architecture independent context switching is applicable to specific classes of problems. Brief assessments of the potential benefits of CSRC technology are given for all of the candidate algorithms explored, along with the rationale for the assessments. Using the CSRC capacity estimates, time domain beamforming is mapped to the CSRC architecture and analyzed. In addition, a CSRC implementation of the optical flow algorithm for plume detection is illustrated. Finally, conclusions are advanced as a result of the candidate algorithms studied.

3.5.2 Candidate Algorithms

Time Domain Beamforming: High Potential

Beamforming is common to a wide variety of sonar and radar applications including the Large Bandwidth Variable Depth Sonar program. The time domain beamforming algorithm filters signals from an array of sensors to amplify signals arriving at the array from a number of specific look directions, or beams. The filtering to form each beam is independent, and consists of simple finite impulse response (FIR) filters that display a symmetry which can be exploited for computational advantage. These properties, along with the fact that the time histories of the sensor data must be shared among all of the beam calculations make the beamforming application particularly attractive for CSRC implementation.

Optical Flow: High Potential

DARPA has identified plume detection as one of its challenge problems. The goal of the plume detection problem is to segment a plume from the background in grayscale video frames. A key algorithm for solving the plume detection problem is the estimation of

optical flow. CSRC FPGAs are well suited for optical flow estimation due to the use of integer arithmetic and the need to access the data in several planes. Additionally, the optical flow estimation requires operations to be performed in serial, which can take advantage of the fast context switching of the CSRC architecture. The calculation of image, spatial and temporal gradients are isolated in our proposed implementation and take full advantage of the CSRC hardware. The efficient evaluation of these core image processing routines may also find application in the Defense Advanced Research Projects Agency (DARPA) challenge problem for text and face recognition in video broadcasts.

Symmetric Polyphase Filter: No Potential

A polyphase filter network can be used to efficiently implement a filter bank as was done for the Common Sensor Interface (CSI) Digital Downconverter (CDDC). The filter performance is limited by the number of filter taps that can be implemented. Each filter tap requires a multiply and in an application specific integrated circuit (ASIC) implementation, such as was used for the CDDC, multipliers are costly in terms of gates. A non-polyphase implementation of a filter can take advantage of filter symmetry to halve the number of multiplies by summing input data prior to multiplying by filter coefficients. The exploitation of the filter symmetry can be expressed as:

$$y(n) = \sum_{l=0}^{L-1} h(l)x(n-l) = \sum_{l=0}^{\frac{L}{2}-1} h(l)[x(l) + x(L-1-l)] \text{ for } h(l) = h(L-1-l).$$

It had been hoped that the polyphase filter network could similarly take advantage of filter symmetry by using reconfigurable interconnects. However, after further investigation it was determined that the filter coefficient reordering used by the polyphase filter network makes exploiting the filter symmetry impossible. The polyphase filter network operation can be expressed as:

$$y_p(n) = \sum_{l=-\infty}^{\infty} h_{n-p}(-l)x_p(l),$$

where

$$h_p(l) = h(lK + p)$$

and

$$x_p(l) = x(lK + p).$$

By dividing the filter, h , into subfilters, h_p , the polyphase filter network can perform the filtering operation once for multiple output channels. Unfortunately, as shown in the example in Figure 3.5.1, the subfilters of a symmetric filter do not have any symmetry. It would be possible to divide a symmetric filter into symmetric subfilters, but the subfilters would not be suitable for use as a polyphase filter bank.

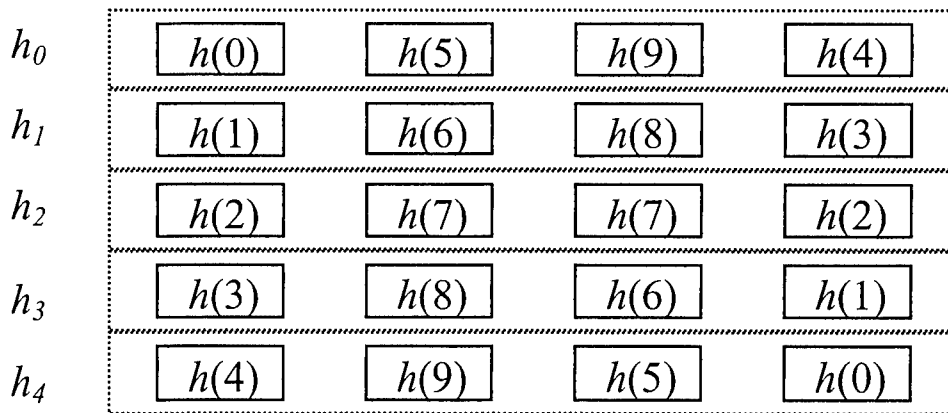


Figure 3.5.1: Polyphase filter network division of symmetric filter into non-symmetric subfilters

Algorithm Cache: High Potential

Current distributed computing architectures like Enhanced Modular Signal Processor (EMSP) and upgrades to the EMSP designed under the RASSP program, typically utilize general purpose processing elements. A high-level scheduler sends algorithms or algorithm segments and the accompanying data to the computing nodes to maximize the throughput of the system. A distributed computing architecture that uses CSRC FPGAs (either alone or in conjunction with general purpose computers) as processing nodes can increase system throughput by evaluating specific algorithms in FPGA hardware rather than on a digital signal processor (DSP). CSRC technology provides a clear advantage over commercial FPGAs since the CSRC devices can retain algorithms in each context layer that is available (effectively caching the algorithms) without imposing the cost of reprogramming an FPGA. Furthermore, context layers not in use can be reprogrammed if a new algorithm not retained in one of the context layers is needed. Finally, if the processing specifies a sequential application of algorithms, the context layers can be set up to share memory locations so that the output of one algorithm is immediately available as the input to the next algorithm (when the context layers are switched). This is not possible with commercial FPGAs. For these reasons, we believe that CSRC technology is very well-suited to distributed computing architectures in the long run. An effective demonstration of this technology includes a significant integration effort with existing schedulers and updating of existing algorithm libraries (e.g., to include word size options). This, along with the fact that the currently available number of gates on a CSRC device limits the size of the algorithms, makes a demonstration of throughput improvement a time consuming prospect.

FOPEN: Medium Potential

Foliage penetrating (FPOEN) target detection in synthetic aperture radar (SAR) images is a simple and natural fit for CSRC technology since the data itself drives the selection of the signal processing algorithm used for target detection. A quick look at the input image determines whether foliage is present, and based upon this test, one of two subsequent target detection algorithms is applied to the image data. The limited number of gates available on CSRC context layers hinders the processing of large images. We expect only a limited improvement over commercial FPGA technology since essentially only

three context layers would be utilized (one to determine which target detection algorithm to use, and for each of the target detection algorithms).

Adaptive Noise Cancellation: Medium Potential

Distributed adaptive noise cancellation is an enabling technology for the SmartSkin demonstration program. Sensors, actuators and signal processing and control electronics are embedded in tiles that cover large surfaces. Coefficients of FIR filters in the controller are adapted during a training period to optimally cancel echoes resulting from incoming acoustic waves. After training, several FIR filters must be evaluated on the locally sensed signals to achieve cancellation. The distributed nature of the problem assigns several sensors and actuators to one processing node carrying out the cancellation control law. A CSRC implementation offers reduced size and the capability to sequence through the processing for several control nodes by switching contexts. Additionally, we may be able to reconfigure the processing node density and FIR filter lengths. This is advantageous because the cancelling of low frequency waveforms typically require longer FIR filters, but a lower processing density (less control nodes per square foot). The implementation details, however, are close to that for beamforming, and the CLB size of the CSRC FPGAs would necessitate the use of external RAM to store the time histories.

Reconfigurable DFT: Low Potential

The CSI radar receiver channelizer is currently using ASIC technology to implement a Winograd discrete Fourier transform (DFT) in hardware. The Winograd DFT algorithm is efficient for computing transforms with lengths that are not an integer power of two, minimizes the number of multiplies, and can be decomposed so that larger sized DFTs can be constructed from smaller sized DFT components. Unlike the fast Fourier transform (FFT) algorithm, the Winograd DFT data flow is complicated and not regular. A more flexible and powerful channelizer is one that could be reconfigured in real time to compute DFTs of varying length. An initial look at applying CSRC technology implementation of a reconfigurable DFT algorithm revealed that complicated switching matrices would be required to handle the data flow, and that even seven context layers per CSRC FPGA yields an unacceptably high device count for the CSI application.

Signal Switching Matrix: No Potential

The channelizer for the All Digital Receiver (ADR) program must select in real time 10 of 28 digital data streams for mixing and low pass filtering and decimation. We see no immediate benefits from the CSRC technology in implementing this switching matrix function over conventional gate logic.

DARPA Challenge Problems Investigated at University of California, Los Angeles (UCLA)

Two DARPA challenge problems have high potential to exploit the advantages of the CSRC architecture: the surveillance challenge problem and the INFOSEC architectures challenge problem. Both problems are being investigated at UCLA and have not been considered for further analysis under this Math Study. The surveillance problem involves the automatic detection and identification of high resolution synthetic aperture radar

(SAR) images. A core algorithm in this application is template matching, which seems quite well-suited to a CSRC architecture that hard wires templates in the context layers for comparison against an image (or subimage) loaded into RAM accessible by all context layers. The INFOSEC architecture problem specifically calls for the development of FPGA hardware that is secure and appropriate for INFOSEC encryption (and related) algorithms, and is currently the focus of attention at UCLA.

3.5.3 Time Domain Beamforming

At the core of sonar and radar signal processing is the notion of beamforming. Incoming waves (acoustic or electromagnetic) are received by an array of sensors. The sensed signals are digitized and processed to determine the direction of the source of the incoming waves. Figure 3.5.2 shows a line array of uniformly spaced sensors in the presence of a point source. The point source is taken to be distant enough so that circular effects are negligible. Thus, when the wavefront arrives at the line array it is planar, and the angle of arrival θ , is measured with respect to the normal of the line array. Under these conditions, the signals on each channel of the line array are related to one another by simple time delays:

$$c_i(t) = c_0(t - \tau_i),$$

where $c_i(t)$ is the continuous time signal sensed on channel i , and

$$\tau_i = i \frac{d}{c} \sin \theta.$$

As shown in the diagram, d is the distance between sensors, θ is the angle of arrival of the incoming wave, and c is the speed of propagation of the wave. This establishes a fundamental functional relationship between the wave's angle of arrival and the time of arrival of the wave at each channel sensor.

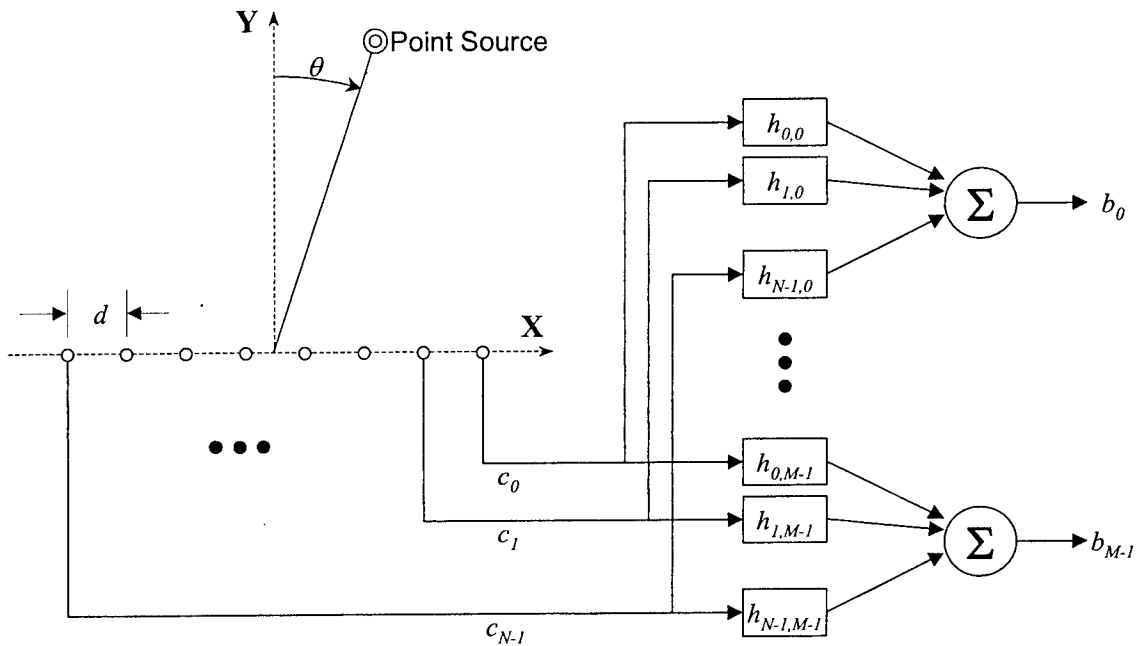


Figure 3.5.2: Time Domain Beamforming

Time domain beamforming passes the channel signals through delay filters (or filters approximating time delays) to temporally align them so that a coherent summation of the delayed channels amplifies signals associated with waves arriving from a particular angle. A coherent summation of N delayed channels that amplifies the signals from the point source in our example of Figure 3.5.2 is:

$$\sum_{i=0}^{N-1} c_i(t - \tau_{N-i-1}).$$

By changing θ , and consequently the delays τ_i , we can change the direction in which the above summation looks. Moreover, as illustrated in Figure 3.5.2, the channel data can be processed in parallel to produce M beams, each with a different look angle. In particular, the j -th beam, which has look angle θ_j , is the coherent summation:

$$b_j(t) = \sum_{i=0}^{N-1} c_i(t - \tau_{N-i-1,j}),$$

where the time delays are given by:

$$\tau_{i,j} = i \frac{d}{c} \sin \theta_j.$$

In Figure 3.5.2, the time delays are approximated by realizable filters $h_{i,j}$, each approximating a delay of $\tau_{N-i-1,j}$.

A digital time domain beamforming algorithm using finite impulse response (FIR) filters $h_{i,j}(n)$ is expressed as:

$$b_j(n) = \sum_{i=0}^{N-1} \sum_{l=0}^{L-1} h_{i,j}(l) c_i(n-l),$$

where n is the discrete time sampling index, T denotes the sampling period, and L is the length of the FIR filters, which must be large enough to account for the longest time delay needed:

$$L > \max_{i,j} \left(\frac{\tau_{i,j}}{T} \right).$$

In practice, the time delay FIR filters will only have K non-zero coefficients which interpolate the appropriate data samples to approximate delays which are not integer multiples of the sampling period. Thus, the evaluation of each FIR filter requires only K multiplies (rather than L). Additionally, the beamforming FIR filters also exhibit channel symmetry:

$$h_{i,j}(n) = h_{N-i-1,j}(L-n-1),$$

which can be exploited to halve the number of multiplies as long as the data from both channels i and $N-i-1$ are available:

$$b_j(n) = \sum_{i=0}^{\frac{N}{2}-1} \sum_{l=0}^{L-1} h_{i,j}(l) (c_i(n-l) + c_{N-i-1}(n+l-L+1)),$$

for even N . Finally, we note that the above symmetry is usually maintained even when the beamforming FIR filter coefficients are shaded with, for example, Taylor windows.

The time domain beamforming algorithm is thus naturally parallel in that the computations for each beam may be done simultaneously. It can be further segmented by evaluating the beamforming equation over subsets of channel indices to generate partial beams, which are then summed to complete the beams. The partial beam for channel i and beam j is:

$$p_{i,j}(n) = \sum_{l=0}^{L-1} h_{i,j}(l) c_i(n-l).$$

Subbeams are formed when partial beams are summed over a subset I of the available channels (rather than over all of the channels, which forms a beam):

$$s_{I,j}(n) = \sum_{i \in I} p_{i,j}(n).$$

The fact that the beamforming expression lends itself naturally to parallel implementations allows for a simple mapping onto the CSRC hardware, which exploits symmetry and identically zero FIR filter coefficients for computational advantage.

Each CSRC FPGA stores N_f channel time histories of length L in configurable logic blocks (CLBs) which are configured as RAM accessible from all context layers. Thus, F CSRC FPGAs can store the time histories for $N = N_f F$ channels. Based on the number of remaining CLBs, each of the C context layers can process M_c subbeams. Thus, a single CSRC FPGA can compute $M = M_c C$ subbeams associated with its N_f channel inputs. Figure 3.5.3 illustrates this decomposition for a single CSRC FPGA, where the input channel vector $\mathbf{c}_{I(i_f)}(n)$ is the set of N_f channel histories $c_i(n)$ for

which $i \in I(i_f)$. The FPGA index i_f thus determines which group of channel histories is presented to each of the CSRC FPGAs, $i_f = 0, 1, \dots, F-1$. In order to exploit the channel symmetry discussed earlier for computational advantage, the number of channel histories stored on each CSRC FPGA, N_f , must be even. Furthermore, channels i and $N-i-1$ must both be contained in one of the groups $I(i_f)$.

A system consisting of F CSRC FPGAs, each receiving N_f channels, can thus process $N = N_f F$ channels into $FM_c C$ subbeams, which are then summed to produce $M_c C$ beams. If a greater number of beams is required, this architecture can simply be replicated (any number of times), with each such system capable of producing $M_c C$ independent beams. Operationally, each CSRC FPGA is presented with the current sample for each of its input channels simultaneously. These inputs are pushed into the time history buffers (tap delay lines) in the first context layer, which also computes the first set of subbeams. The FPGA then sequences through the remaining context layers to produce the rest of the subbeams. Each context layer sends its output to the same set of I/O pins, so that the subbeams are multiplexed on these lines.

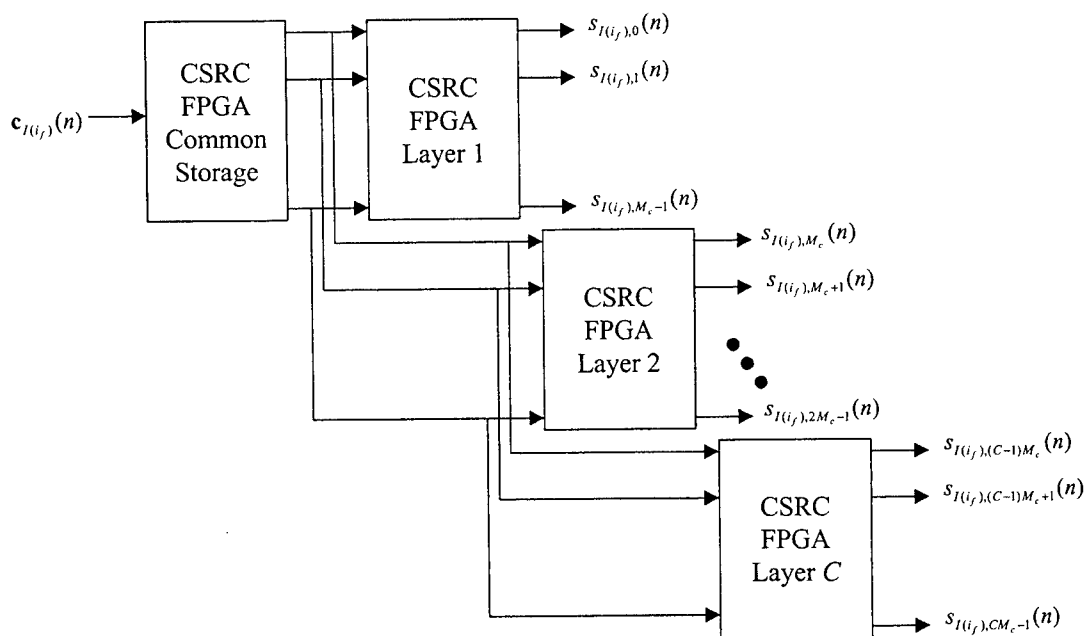


Figure 3.5.3: Subbeam decomposition onto CSRC FPGA

A slight variation to the architecture described above is to push new samples into the channel history buffers in all context layers (not just the first context layer). Each beam would thus be computed every C samples, (but the samples are now arriving C times faster). The advantage of this architecture is that the control lines which specify the active context layer, and hence the output beams, can be commanded in any order. This provides a hardware device that fits nicely into typical sonar and radar sensing operations which command a search pattern (e.g., scanning through all the beams) until a target is detected, and then track the target by selecting beams immediately around it.

Table 3.5.1 performs sizing calculations for a single context layer of a CSRC FPGA for the beamforming application. Using [LR1] two input channels and two subbeams per context layer keeps the implementation below 80 percent utilization since each context layer has 1024 CLBs in the current Sanders design. The calculations in the table include taking advantage of the beamformer channel symmetry, and the savings that accrue from configuring multipliers where one operand (the FIR filter coefficient in this case) is known a priori and is hard wired in the context layer.

We can compare a beamforming system utilizing CSRC FPGA with alternative systems using standard FPGA components. For these comparisons, we assume that a standard FPGA has the same number of CLBs as a CSRC FPGA context layer of the same area, but that a standard FPGA CLBs is twice as capable as a CSRC CLB. One alternative system essentially uses one standard FPGA to compute what we have allocated to a CSRC FPGA context layer. This alternative approach consumes $\frac{1}{2}C$ times as much volume as the CSRC implementation. A second alternative is to replace each CSRC FPGA with one standard FPGA which clocks through the subbeam calculations. This approach suffers from the following inefficiencies compared to the CSRC FPGA implementation: (1) the FIR filter coefficients must be stored (in the CSRC they are hard wired), consuming RAM, (2) the multipliers must be general, (3) the operands of the

Number of Channels	2	2
Number of Subbeams per Context layer	3	2
Length of Channel Histories	256	256
Length of Beamforming FIRs	4	4
Channel and FIR Coefficient Data Width (bits)	8	8
Subbeam Data Width (bits)	16	16
CLBs per RAM bit	0.0625	0.0625
CLBs per Channel Data Width Multiply	40	40
CLBs per One Bit Addition	1	1
CLBs allocated for RAM	256	256
CLBs allocated for Multiplies	480	320
CLBs allocated for Additions	288	192
Total CLBs Required	1024	768

Table 3.5.1: CSRC Context Layer Sizing For Beamforming

multipliers must each be loaded into operand registers, and (4) the indexing to find the correct samples of the time histories that must be loaded into the operand registers must be implemented in the logic (the CSRC implementation hardwires the correct locations to a multiplier dedicated to a single FIR coefficient). All of these inefficiencies mean that this standard FPGA approach would not be able to compute as many beams as the equivalent CSRC FPGA implementation. In fact, accounting for just the first two effects listed above makes the standard FPGA and the CSRC FPGA approximately equal in capability. The logic associated with loading operands into registers and indexing into the channel time histories is avoided in the CSRC implementation and the performance gap widens as the number of context layers per CSRC FPGA increases.

Our conclusion from this comparison, is that CSRC provides a better approach to time-domain beamforming than conventional FPGAs, resulting in smaller volume for a specific number of beams. To demonstrate the data flow in a CSRC beamforming application, a MATLAB/SIMULINK computer simulation has been developed. The simulation assumes seven context layers, each large enough to permit four input channels and four subbeams, per CSRC FPGA. The numbers of input and output channels per CSRC FPGA are hardwired in SIMULINK, as is the upper limit of seven for the number of context layers per FPGA. Changing these quantities requires block diagram editing of the simulation, and a small amount of text editing of the initialization m-file (which is annotated in the places where editing will be required). The input to the simulation is an idealized set of channel time histories seen by a line array in response to a moving point source in the far field. Easily modifiable parameters specify the straight line motion of the point source, the line array configuration and CSRC properties not listed as hardwired above. Figure 3.5.4 shows the windowed energy of the time histories of all 28 beams in the simulation for a point source moving parallel to the X direction with constant velocity. The beam containing the maximum energy (brighter pixels represent larger energy) changes as the point source moves. As expected, the beam corresponding to the angle the point source makes with the normal to the line array exhibits the highest energy throughout the simulation.

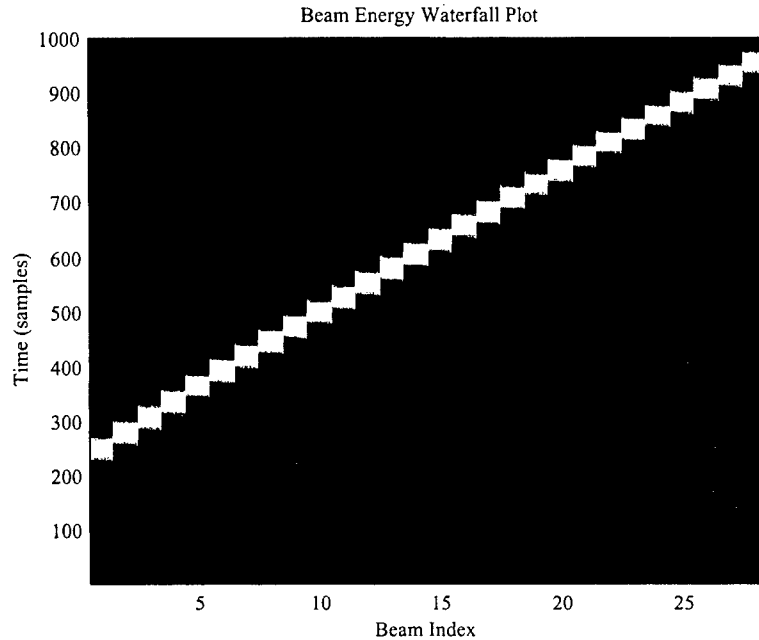


Figure 3.5.4: SIMULINK CSRC Beamforming Simulation Example Output

3.5.3 Plume Detection

The goal of the plume detection DARPA challenge problem is to segment a plume from the background in gray scale video frames. The proposed approach is to assume that the plume is the only moving object in the frames. Using an optical flow algorithm to estimate velocities within the frame, the plume can be segmented by thresholding the estimated velocities. The challenge problem seeks architectures which can process 256 by 256 images with 12-bit pixels at a frame rate of approximately 1 kHz.

Optical Flow Algorithm Description

Using the assumption of constant illumination, the optical flow algorithm considers any changes in pixel intensity in time to be caused by object motion. The intensity change due to object motion can be expressed as:

$$I_t = -u(x, y, t)I_x - v(x, y, t)I_y, \quad [3.6.3-1]$$

where I is the image intensity, u and v are the x and y velocities of objects within the image, $I_t = \partial I(x, y, t) / \partial t$, $I_x = \partial I(x, y, t) / \partial x$, and $I_y = \partial I(x, y, t) / \partial y$. Equation 3.6.3-1 does not account for second order effects at moving object boundaries. Employing estimates \hat{u} and \hat{v} for the unknown object velocities, a measure of the accuracy of the estimates is

$$e(x, y, t) = I_t + \hat{u}(x, y, t-1)I_x + \hat{v}(x, y, t-1)I_y,$$

where e is the residual of the estimated change in intensity with respect to time. If the image is divided into subimages, the total error can be expressed as

$$E = \sum_{\text{subimage}} e(x, y, t).$$

The subimage error, E , can then be used to update the velocity estimates using the relations

$$\hat{u}(x, y, t) = \hat{u}(x, y, t-1) - bEI_x$$

and

$$\hat{v}(x, y, t) = \hat{v}(x, y, t-1) - bEI_y,$$

where b is a learning parameter. The choice of b and subimage size affect the performance and stability of the optical flow algorithm.

FPGA Architecture

The ability of the CSRC FPGAs to retain memory during context switching makes the FPGAs well suited for computing the partial derivatives of the optical flow algorithm. By loading in data from successive frames, a single FPGA can compute I_t , I_x and I_y using one layer for each partial derivative. The partial derivative data can then be passed to a second FPGA to perform the error calculation and velocity estimate updates. Figure 3.5.5 shows a block diagram of a processing block containing two CSRC FPGAs with a shared 128 Kbyte memory. Both FPGAs have access to the RAM and the two FPGAs have data and control connections between themselves. The connections for the FPGAs and RAM are summarized below:

FPGA1 (256 I/O pins)

- 80 pins connected to FPGA2
- 80 pins connected to external RAM
- 16 pins address, 64 pins data
- 16 pins external control

FPGA2 (256 I/O pins)

- 80 pins connected to FPGA1
- 80 pins connected to external RAM
- 16 pins address, 64 pins data

RAM

- 16 pins address
- 64 pins data
- control pins

The 64-bit data connections allow for 4 words to be processed in parallel. A mechanism for arbitration of the RAM bus between the FPGAs and external access would be required. Use of dual-port RAM could simplify the arbitration and improve performance.

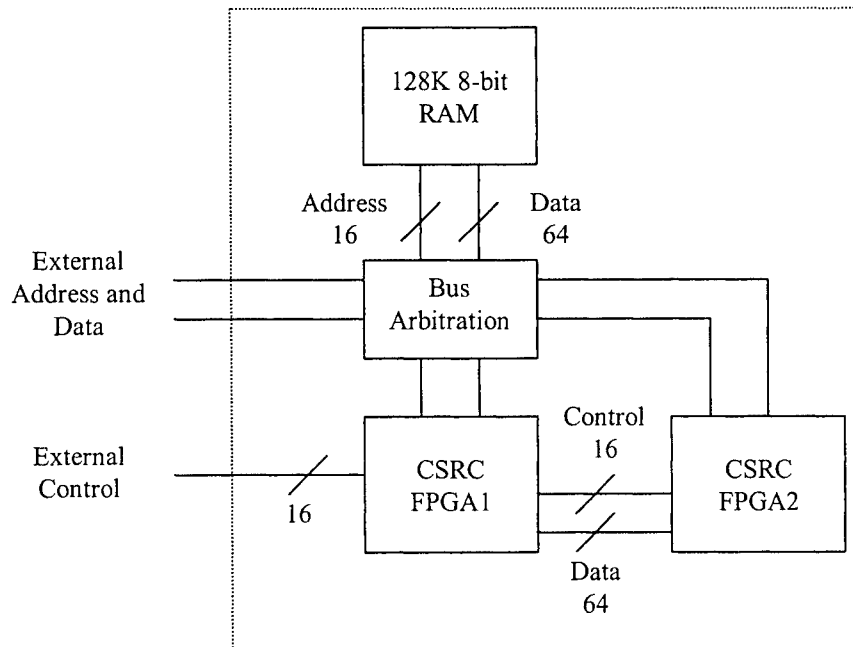


Figure 3.5.5: CSRC Plume Detection Processing Block Architecture

The sizing of the plume detection problem discussed in later sections indicates that eight of the processing blocks shown in Figure 3.5.5 would be necessary for 256 by 256 pixel images at a frame rate of 1 kHz. Figure 3.5.6 shows a CSRC architecture block diagram containing eight processing blocks, each with external data and control connections. No communication between the processing blocks would be necessary. The following sections discuss the mapping of the plume detection problem to the architecture shown in Figure 3.5.6 and the corresponding resource allocation.

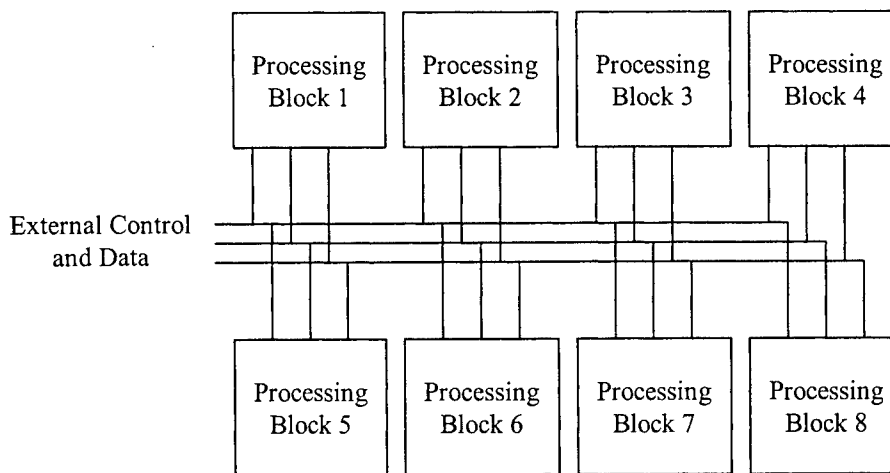


Figure 3.5.6: CSRC Plume Detection Architecture

Algorithm Mapping

The plume detection problem can be implemented on the CSRC architecture shown in Figure 3.5.6 by having the processing blocks operating in parallel on different subimages. Dividing the 256 by 256 image into sixteen 64 by 64 pixel subimages allows each processing block to operate upon two subimages. Each processing block processes its

two subimages serially. The subimage data would be loaded into the processing blocks' RAM from the external interface. The external interface would step through the processing blocks reading the updated velocity estimates and writing the next video frame for the appropriate subimage. Given proper RAM bus arbitration, the subimages could be double buffered to allow the next frame to be loaded while the FPGAs operate upon previous frames. A similar double buffering approach could be used for the output velocity estimates.

Table 3.5.2 describes the mapping of the plume detection problem for a single subimage to the CSRC processing block shown in Figure 3.5.5. Nine processing steps are listed in Table 3.5.2 in the order in which they would be performed and the corresponding operations performed by each FPGA are described. Each step operates upon four words at a time. The two FPGAs have been designated FPGA1 and FPGA2. In step 1, FPGA1 reads two frames of a 16 by 16 partial subimage into the FPGA local memory using its first context layer. For the most recent frame some extra pixels would need to be read to compute I_x and I_y . FPGA1 then switches to context layer 2 for step 2, computes I_x and sends I_x to FPGA2. FPGA2, using its first context layer, receives I_x , multiplies I_x by \hat{u} and begins the summation for E . For step 3 FPGA1 switches to context layer 3 to compute I_y and FPGA2 continues to use layer 1 to add $\hat{v} I_y$ to E . FPGA1 then switches to layer 4 to compute I_t for step 4 and FPGA2 switches to layer 2 to add I_t to E . Steps 2 through 4 are repeated for each partial subimage until the entire subimage has been processed and the E summation is complete. When the E summation is complete, FPGA layer 2 computes the product bE used for velocity estimate updates. For step 6, FPGA1 reloads context layer 1 and reloads the most recent frame of the partial subimage. FPGA1 then computes I_x and I_y in steps 7 and 8 the same as in steps 2 and 3 and FPGA2 switches to layer 3 to use I_x and I_y to update the velocity estimates which are stored in RAM. FPGA2 could use its fourth layer following step 8 for additional functions such as velocity thresholding or grayscaling. Steps 6 through 8 are repeated until the velocity estimates have been updated for the entire subimage. The FPGA and memory resource utilization for this mapping of the optical flow algorithm are discussed in the following sections.

Step	FPGA1		FPGA2	
	Layer	Operation	Layer	Operation
1	1	Read partial subimage (16x16) $I(x,y,t)$ and previous frame partial subimage $I(x,y,t-1)$ from RAM		
2	2	Compute I_x and output I_x to FPGA2 (4 pixels at a time)	1	Compute $\hat{u} I_x$ and keep a running sum of E
3	3	Compute I_y and output I_y to FPGA2 (4 pixels at a time)	1	Compute $\hat{v} I_y$ and keep a running sum of E
4	4	Compute I_t and output I_t to FPGA2 (4 pixels at a time)	2	Add I_t to E and compute bE product
5	Repeat steps 1 through 4 to complete subimage			
6	1	Read partial subimage (16x16) $I(x,y,t)$ from RAM		
7	2	Compute I_x and output I_x to FPGA2 (4 pixels at a time)	3	Update \hat{u}
8	3	Compute I_y and output I_y to FPGA2 (4 pixels at a time)	3	Update \hat{v}
9	Repeat steps 6 through 8 to complete subimage			

Table 3.5.2: CSRC Optical Flow Algorithm Mapping

Processing Requirements

The processing requirements for implementing the optical flow algorithm as discussed above are examined in this section in terms of the number of FPGA clock cycles required. Table 3.5.3 lists the number of FPGA clock cycles necessary for each step of the optical flow algorithm. Each 64-bit RAM access is assumed to take two clock cycles. Most of the steps operate in a pipelined fashion where 4-word vectors are pumped through the pipe in parallel. The number of clock cycles required for the pipeline operation consists of two factors. The first factor is a one time cost for each algorithm step which accounts for the entire length of the pipeline. The second factor is the cost for the slowest part of the pipeline for each 4-word operation. For step 1, FPGA1 reads the most recent subimage frame and the previous frame into memory. Taking into account the 2-cycle, 64-bit reads and the extra data required for the I_x and I_y calculations, step 1 requires 298 cycles for each subimage. In step 2 the slowest part of the pipeline is the calculation of $\hat{u} I_x$. The FPGA only has enough resources for two multipliers, therefore the 4-word multiplication must be done in two steps of two multiplications. With a total pipeline length of 8 cycles and a per vector cost of 2 cycles, step 2 requires 136 cycles for each partial subimage. Functionally, step 3 is identical to step 2. Step 4 does not require multiplies and therefore is assumed to consume 1 cycle per vector with an overhead of 4 cycles and an extra 4 cycles for computing the bE product. The cycle cost of step 5 is the partial subimage cost of steps 1 through 4 times the 16 partial subimages which compose a subimage. Step 6 is similar to step 1 except that the previous frame is not necessary since it does not need to be computed. The memory accesses for reading and writing the velocity estimates are the slowest parts of the pipeline for steps 7 and 8. With two memory accesses at 2 cycles per access the per vector cost is 4 cycles. Step 8 computes the cost of steps 6 through 8 for an entire subimage. The total image cost is then computed based upon the subimage costs for steps 5 and 9. Given an 80 MHz FPGA and

a 1000 Hz frame rate, 80000 cycles are available per processing block for each frame. Eight processing blocks would provide 640000 cycles per frame which would require each processing block to operate at 54 percent of its estimated capacity. The relatively low estimated utilization allows for errors in estimates, slower FPGA parts or added functionality such as velocity thresholding.

Step	Cycles		Notes (all steps operate on 4 words at a time, memory accesses are assumed to take 2 cycles)
1	$C_1=298$	$(17 \times 5 + 16 \times 4) \times 2$	Most recent frame requires 17×17 to compute I_x and I_y
2	$C_2=136$	$(16 \times 4) \times 2 + 8$	Pipelined operation with 8 cycles of setup and 2 cycles per 4-word vector (2 multipliers)
3	$C_3=136$	$(16 \times 4) \times 2 + 8$	Pipelined operation with 8 cycles of setup and 2 cycles per 4-word vector (2 multipliers)
4	$C_4=68$	$(16 \times 4) \times 1 + 8$	Pipelined operation with 4 cycles of setup, 1 cycle per 4-word vector and 4 cycles for bE product
5	$C_5=10208$	$16 \times (C_1 + C_2 + C_3 + C_4)$	64×64 subimage made up of 16 partial subimages
6	$C_6=170$	$(17 \times 5) \times 2$	Only most recent frame required
7	$C_7=272$	$(16 \times 4) \times 4 + 16$	Pipelined operation with 16 cycles of setup and 4 cycles (1 read and 1 write) per 4-word vector
8	$C_8=272$	$(16 \times 4) \times 4 + 16$	Pipelined operation with 16 cycles of setup and 4 cycles (1 read and 1 write) per 4-word vector
9	$C_9=11424$	$16 \times (C_6 + C_7 + C_8)$	64×64 subimage made up of 16 partial subimages
Image total	346112	$16 \times (C_5 + C_9)$	256x256 image made up of 16 subimages

Table 3.5.3: CSRC Optical Flow Processing Requirements

FPGA Resource Requirements

Estimates of the number of FPGA CLBs used for each FPGA layer are discussed in this section. Table 3.5.4 lists the number of CLBs used for each layer for memory and for arithmetic functions. When a CLB is used as memory, the CLB is assumed to provide 16 bits of storage. Implementing an adder in the FPGA is presumed to require one CLB for each bit in the adder. For example, a 12-bit adder would require twelve CLBs. A general coefficient multiply, where both multiplicands are variable, consumes an estimated 300 CLBs. For the first layer of FPGA1 no arithmetic functions are performed, but CLBs are required to store the partial subimage frames. FPGA1 layers 2 through 4 maintains the memory loaded by layer 1 and each require 4 12-bit adders. Layer 1 of FPGA2 needs two 12-bit multipliers for the $\hat{u} I_x$ or $\hat{v} I_y$ calculation, four 24-bit adders for the E summation and 24 bits of storage for E. FPGA2 layer 2 has the same requirements as layer 1 without the multipliers. Without the need for the same multipliers as layer 1, layer 2 is a good place to perform the bE multiplication at the end of a subimage. The bE multiplication can be done as a constant coefficient. Given b is a 12-bit constant and E is 24 bits, the bE multiplication would consume 200 CLBs. The third layer of FPGA2 requires two 12-bit multipliers and two 16-bit adders for the velocity estimate updates and 12-bits of storage for bE . The maximum number of CLBs used by any FPGA layer is 698. With an FPGA containing 1024 CLBs there should be sufficient room to allow for control logic, programming overhead and possible expansions of number of bits used

in the arithmetic. Increasing the number of bits used in the arithmetic could reduce overflow and underflow problems.

FPGA	Layer	Required CLBs		
		Total	Memory (16 bits per CLB)	Computational (1 CLB per bit add, 300 CLB per 12 bit by 12 bit general coefficient multiply)
1	1	545	$17 \times 17 + 16 \times 16$	0
1	2	593	$17 \times 17 + 16 \times 16$	12×4
1	3	593	$17 \times 17 + 16 \times 16$	12×4
1	4	593	$17 \times 17 + 16 \times 16$	12×4
2	1	698	2	$300 \times 2 + 24 \times 4$
2	2	298	2	$200 + 24 \times 4$
2	3	633	1	$300 \times 2 + 16 \times 2$
2	4	N/A		

Table 3.5.4. CSRC Optical Flow Algorithm FPGA Resource Allocation

Memory Requirements

Table 3.5.5 lists the RAM requirements for the CSRC optical flow architecture. The optical flow algorithm stores the frame data, I , and the velocity estimate data, \hat{u} and \hat{v} , in the RAM. Additional memory is assumed to be used for double buffering the input frame data and the output velocity estimates. The total memory requirement for each frame is 896 KB. Each of the 8 processing blocks having 128 KB provides a total of 1 MB.

Data	Size (bytes)	
$I(x,y,t+1)$	131072	$2 \times 256 \times 256$
$I(x,y,t)$	131072	$2 \times 256 \times 256$
$I(x,y,t-1)$	131072	$2 \times 256 \times 256$
$\hat{u}(x,y,t-1)$	131072	$2 \times 256 \times 256$
$\hat{u}(x,y,t-1)$	131072	$2 \times 256 \times 256$
$\hat{v}(x,y,t)$	131072	$2 \times 256 \times 256$
$\hat{v}(x,y,t)$	131072	$2 \times 256 \times 256$
Total	917504	

Table 3.5.5. CSRC Optical Flow Algorithm RAM Requirements

CSRC FPGA Alternative Architectures

Alternatives to the CSRC FPGA optical flow architecture include ASIC, integer DSP and standard reconfigurable FPGA architectures. Table 3.5.6 lists the possible optical flow architectures and their respective advantages and disadvantages. The CSRC FPGA architecture implements the 800 Mops optical flow algorithm in real-time while maintaining flexibility for changes to the algorithm. A disadvantage of the CSRC FPGA architecture is that programming of the algorithm would be more difficult than for a general purpose processor such as an integer DSP. An ASIC implementation of the optical algorithm could provide the fastest processing speed along with being power and volume efficient, however an ASIC architecture would have a large up front cost for design and would not be flexible in regards to changes in the algorithm. Using an integer

DSP for the optical flow algorithm would allow for the easiest and most flexible programming, however an integer DSP would only provide about 20 Mops. The integer DSP architecture could achieve the required 800 Mops but would need 40 DSPs, which would be costly in terms of hardware and volume. Two possible approaches exist for using a standard reconfigurable FPGA architecture for the optical flow algorithm. One approach would be to utilize one standard FPGA for each CSRC FPGA layer. Using this approach the speed and flexibility of the CSRC FPGA architecture could be achieved, but the hardware and volume costs would be increased by a factor of over three. Alternatively, one standard FPGA could be used for each CSRC FPGA. Since reprogramming a standard FPGA takes approximately 30 ms, which corresponds to 30 frames at a 1 kHz frame rate, reprogramming could not be used in the 1-to-1 FPGA replacement architecture. Without the ability to reprogram, a different mapping of the optical flow algorithm to the FPGAs where the different functions are distributed across FPGAs would be needed. Spreading the functions across FPGAs complicates the data flow and likely creates memory access bottlenecks. The CSRC FPGA architecture provides a combination of speed and flexibility for the optical flow algorithm which cannot be matched by other architectures.

Architecture	Advantages	Disadvantages
CSRC FPGA	<ul style="list-style-type: none"> • Fast • Flexible 	<ul style="list-style-type: none"> • Moderately difficult programming
ASIC	<ul style="list-style-type: none"> • Very fast • Power and volume efficient 	<ul style="list-style-type: none"> • Very difficult programming • Expensive hardware • Not flexible
Integer DSP	<ul style="list-style-type: none"> • Very flexible • Easy programming 	<ul style="list-style-type: none"> • Slow
standard reconfigurable FPGA (1 FPGA for each CSRC FPGA layer)	<ul style="list-style-type: none"> • Fast • Flexible 	<ul style="list-style-type: none"> • Moderately difficult programming • Expensive hardware • Volume costly
standard reconfigurable FPGA (1 FPGA for each CSRC FPGA)	<ul style="list-style-type: none"> • Flexible 	<ul style="list-style-type: none"> • Slow • Difficult implementation

Table 3.5.6. Optical Flow Algorithm Architecture Comparison

3.5.4 Conclusions

A number of signal processing algorithms have been assessed with regard to their potential to exploit the CSRC hardware architecture for computational advantage. Generally, it seems that the algorithms benefiting the most from the CSRC architecture are those that share a significant number of memory locations between context layers. These early conclusions were one of the driving forces for implementing block RAM in the CSRC FPGA.

Two promising candidate algorithms, time domain beamforming and optical flow, have been mapped onto the CSRC hardware at a functional level, sizing the implementations according to the anticipated capabilities of CSRC context layers. For each of these two

applications, the computational advantages of a CSRC implementation over commercial FPGAs have been identified. In both applications, greater capability in each context layer would increase the computational efficiency. For the time domain beamforming, template matching, and adaptive noise cancellation applications, a larger number of context layers would also increase efficiency, although this is not true for the optical flow algorithm. For the time domain beamforming application, a MATLAB/SIMULINK simulation has been produced to illustrate the operation and data flow of the algorithm as it would be mapped onto CSRC FPGAs.

3.6 Architecture Design

The final architecture as described in section 3.1 came about as a collaborative, iterative process with input from experts including users and developers in industry, academia and in the government. Unlike traditional FPGA development, “share holders” were invited to participate early and often in the design process. In fact, one of the first undertakings of this effort was to solicit desired features from as many experts and consumers of FPGAs (including, but by no means limited to, DoD, JPL, NASA, Lucent, AFRL, USC/ISI, and the entire DARPA ACS community). As a result of this solicitation, Sanders was able to compile a “feature wish list”. It was from this list that the architecture sprung. It should be noted that the loss of Xilinx as the IC designer actually had a positive effect on the overall architecture of the CSRC device. In particular Sanders was able to add features, such as the many security features desired by DoD which would not have been possible with existing commercial design methodologies. Since feature sets for commercial devices is always driven by customer demands and needs that drive commercial volume. Since the security community does not represent a significant portion of Xilinx’s revenue, it is highly unlikely that such features would have been included in the CSRC design.

Another unique input for the architecture was from the tool developer, FPGA technologies. Unlike traditional FPGA designers who design the IC in a vacuum and then throw it over the wall to the software engineers to “figure out” how to place and route the device, Sanders worked closely with the tool developer throughout the architecture design process. Through a series of experiments Sanders was able to derive the optimum architecture (logic, routing, etc.) that also afforded reduced complexity in placement and routing. In fact, benchmark circuits were placed and routed to determine if the architecture was sufficiently flexible and powerful. Simulating the placement and routing, and iterating through the architecture design concluded with an architecture that is not only flexible and sufficiently capable to support a wide range of DSP and glue logic but affords rapid place and route. Reduced place and route times were a goal of this program as it is directly coupled to the cost of the overall design and mapping.

3.7 Prototype Integrated Circuit (IC) Design

Subsequent to Xilinx's departure from the team Sanders decided to determine if it was possible to leverage any of the work being conducted at the University of California, Berkeley. A technical discussion took place between Sanders and UC Berkeley. Both the existing logic cell and the 3-input LUT cell under development were discussed in detail. Based on the meeting, Sanders did some conceptual design of the CSRC device architecture based upon the existing Berkeley cells. However, in the end, the cells were not applicable to the design process and architecture chosen so it was decided to develop the needed cells from scratch. It was also decided that National Semiconductor's I/O cells, which were provided to Sanders, were sufficient for the job and afforded a proven design for ESD protection. In the end, Sanders wrapped control and flexibility around the core I/O cells to form the I/O for the CSRC device. Design, simulation, and signal integrity studies were performed for the output portion of I/O cell. Even though it was not expected that the device would be run at 500MHz, the I/O cell design was fully tested at this speed. Although many of these early design considerations were ultimately discarded, a short discussion of some of the preliminary design ideas follows for completeness.

Figure 3.7.1 shows a block diagram of the first conceived programmable input/output buffer. The pull-down(s) at the pad were not shown because the output of the input buffer can be configured to be either true or inverted. Hence a single pull-up can yield either a logic zero or logic one at the output of the input buffer. It was expected that the programmable input/output buffer would be able to drive and receive TTL/CMOS/LVCMOS logic levels. A programmable delay element was included in the output buffer to facilitate high speed and high skew data input transfers. The two flip-flops in the programmable input/output buffer can receive a clock from one of two sources, however, increasing this to one of four sources was considered.

A schematic of the programmable output buffer and the circuitry required for programming this element are depicted in Figure 3.7.2. The following blocks would have been the memory cells required to hold programming information:

- INV – Output polarity
- SD0 – Drive/Tri-State Select Bit
- SD1 – Drive/Tri-State Select Bit
- SR – Slew Rate Control Bit

After significant architecture refinements, the design of the prototype CSRC IC ensued. Figure 3.7.3 depicts the floorplanned layout of the logic cell and the entire prototype chip (excluding a control block and the I/O), respectively.

Figure 3.7.4 is a photograph of the CSLC Layout. It can be seen that the CSBits (Context Switching Bit – the memory element used to store configuration bits) in the CSLUT

(Context Switching Look-Up Table) dominate the area. These can be seen as the dark and dense vertical strip on the right side of the picture. Figure 3.7.5 is a depiction of the decoders & switchbox layout from the Level 1 routing. This picture includes a decoder, with 4 associated CSBits. Each decoder drives a switch box that connects a single CSLC input to one of the available inputs. Notice that the height of a single switch box closely matches the height of a decoder, thereby allowing for regular placement. Note that the switch box signal distribution design avoids the use of diagonal lines which in turn saves a tremendous amounts of area.

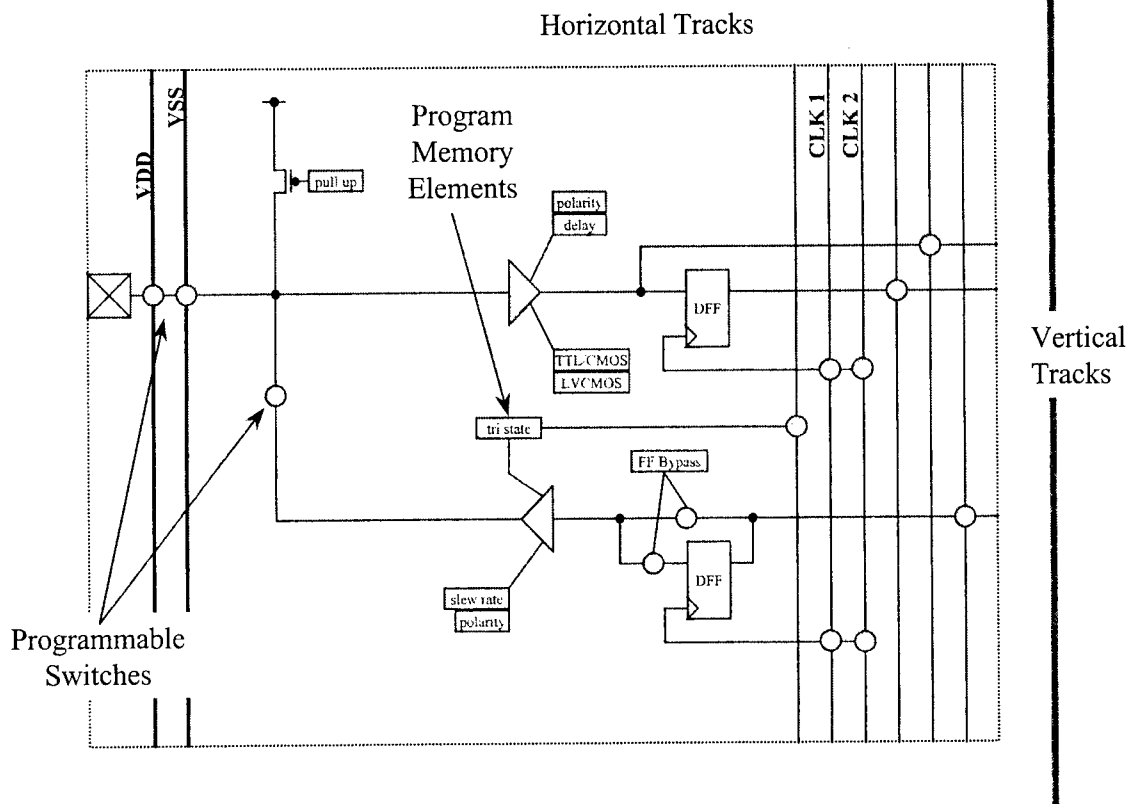


Figure 3.7.1: Block Diagram of I/O Buffer

Transistor Sizing,
output transistors,
all lengths = 0.8u

Device	W
mp1	83
mp2	160
mp3	320
mp4	320
mps1	84
mps2	166
mps3	167
mn1	33
mn2	70
mn3	140
mn4	140
mns1	34
mns2	66
mns3	67

Performance
(Cloud=10pF,
Vdd=3.3V
Temp=70C)

Max Data Rate
200 Mhz

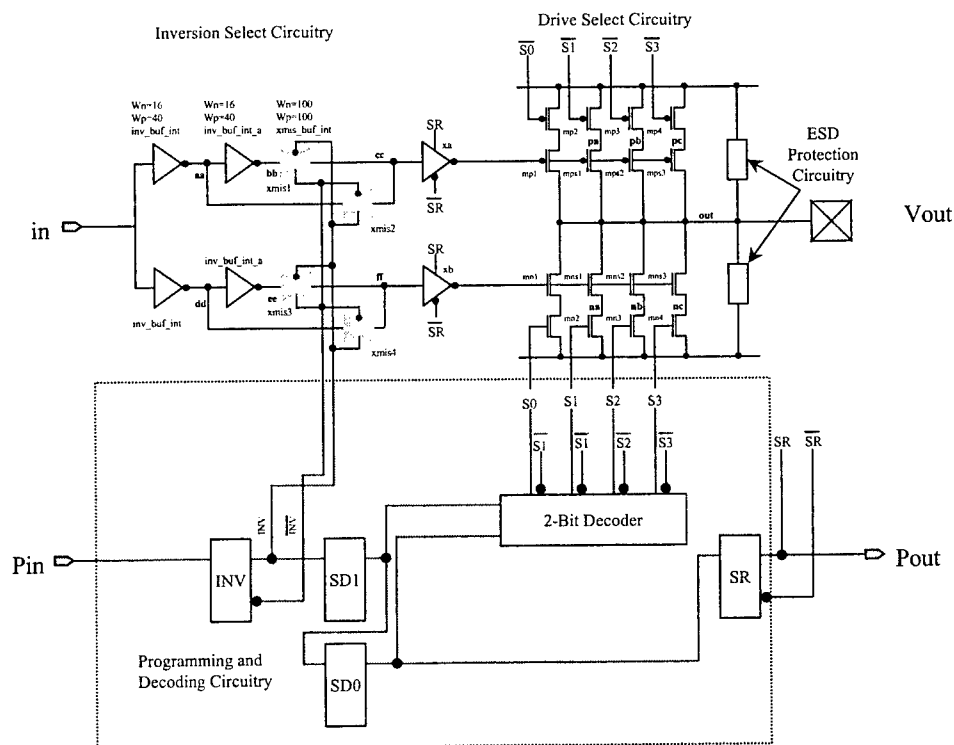


Figure 3.7.2: Programmable Output Buffer

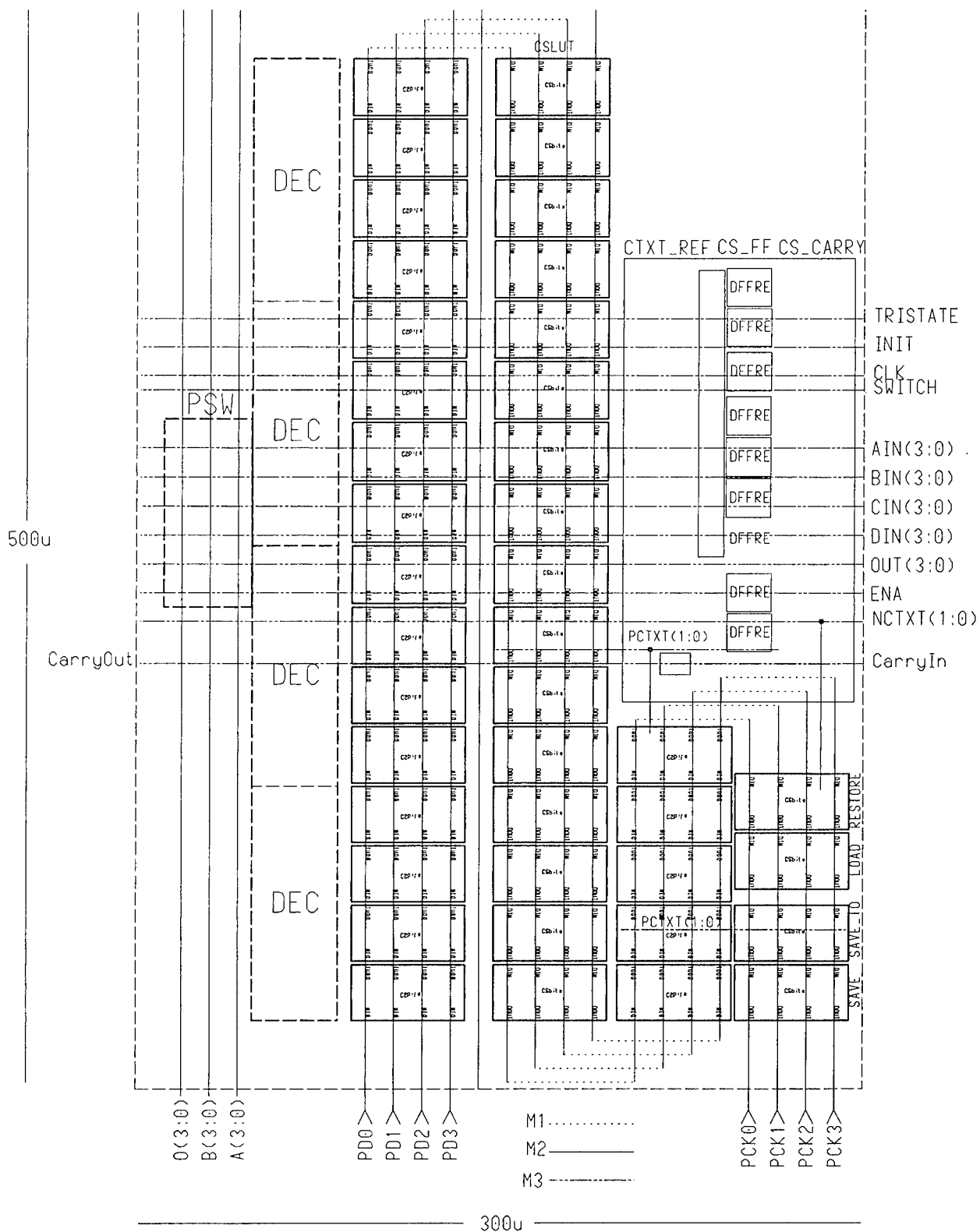


Figure 3.7.3: CSLC (Logic Cell) Floorplan

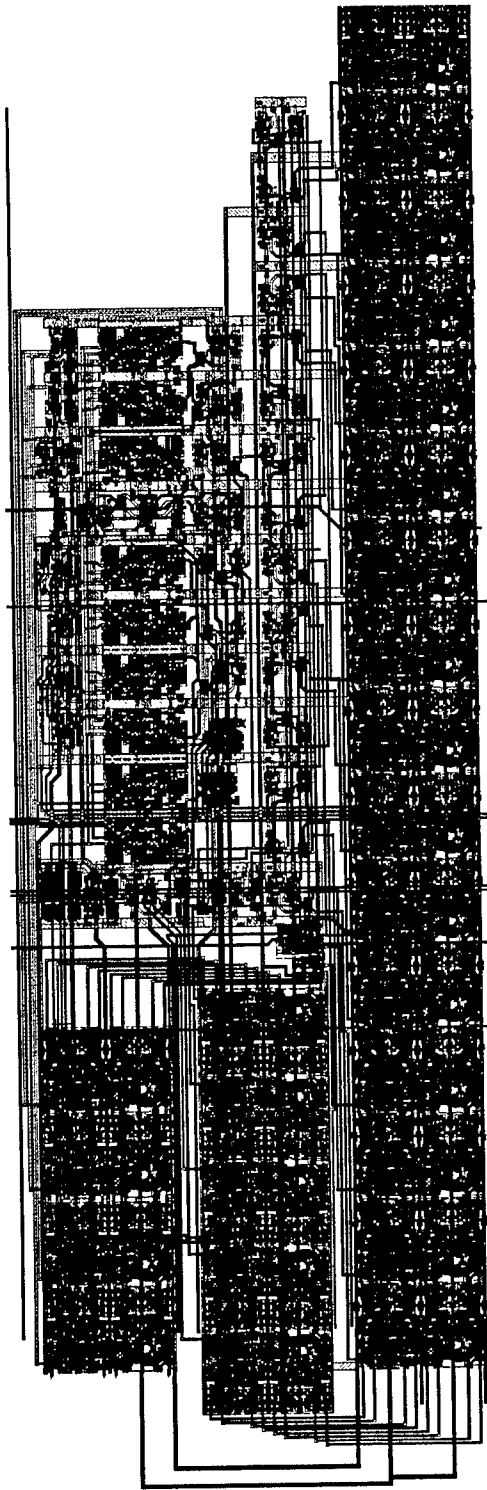


Figure 3.7.4: CSLC (Logic Cell) Layout

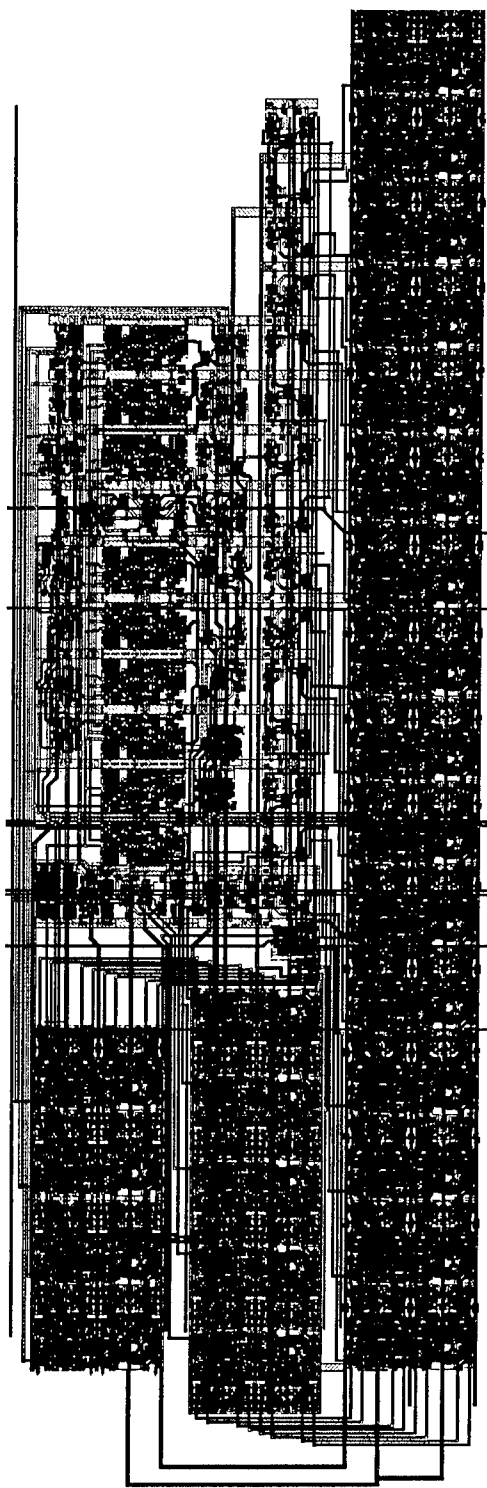


Figure 3.7.5: Decoders and Switchbox Layout (Level 1 Routing)

With the above described building blocks, the CSLC was integrated with the decoders for the first level of routing (matrix level 1 routing) and the routing itself. The global connections between 4 CSLCs were then defined and laid out. Following the completion of this effort, the layout of a logic array (16 CSLC's) with their higher level connections was completed. All layouts were automatically checked for consistency between layout and schematic (LVS). Recall that the I/O cells were taken from National Semiconductors library because it was decided that this solution provided the minimum risk since it would not only save time but would guarantee successful implementation of the electrostatic discharge (ESD) protection. Figure 3.7.6 shows the layout of the entire CSRC prototype chip without the control block.

The integrated circuit design effort for the CSRC prototype concluding with the tape-out to National Semiconductor. Note that Sanders sent a GDSII database to National Semiconductor who created the masks, fabricated the IC on their .35u silicon process, and packaged the devices. The attached picture (Figure 3.7.7) is the final layout of the entire prototype CSRC device.

The highlight of the prototype IC design effort was the return of the CSRC prototype FPGA. The chip was exercised on a Topaz IC tester at Sanders and was proven to be functioning as designed. The initial testing of the prototype included programming a context, switching to a configured context, and running data through the active context. Continued testing of this device is ongoing on the IC tester. This first pass success of the 500,000+ full-custom transistor design of the world's first context switchable FPGA is credited to the thoroughness with which the design was done. Some of the steps taken to ensure success (in addition to the VHDL structural modeling) were rooted in basic simulation & verification and consisted of:

1. Complete schematic simulation of low level and mid level blocks using HSPICE,
2. Verifying the integrity of the layout using the foundry supplied design rules (DRC)
3. Verification of the correctness of the layout using layout vs. schematic tool (lvs)
4. Verification of the behavior of the architecture based on VHDL models of the low level cells

The simulations were done on each cell and block continuously during the design and layout of the chip. Prior to tape-out, the behavior of the entire chip was successfully simulated, including context switching, using the functional VHDL models of the lowest level schematic cells and the Perl scripts. Large critical blocks of layout (up to about 2500 transistors) were functionally verified using HSPICE to further test the electrical integrity of the design. Finally, the layout of the whole chip matched its schematic description.

Finally, Sanders included test structures of the most critical blocks of the prototype on the retical in order to obtain details about internal waveforms and to obtain information about

the capabilities of the 0.35 micron process from National Semiconductor. The test structures included the control block, the various memory units, routing elements, I/O cells, large drivers, logic cells, and context programming units. All test structures were simulated and verified to conform to the schematics. Note that due the success of the overall CSRC prototype IC, the test structures were fortunately not required to debug the device.

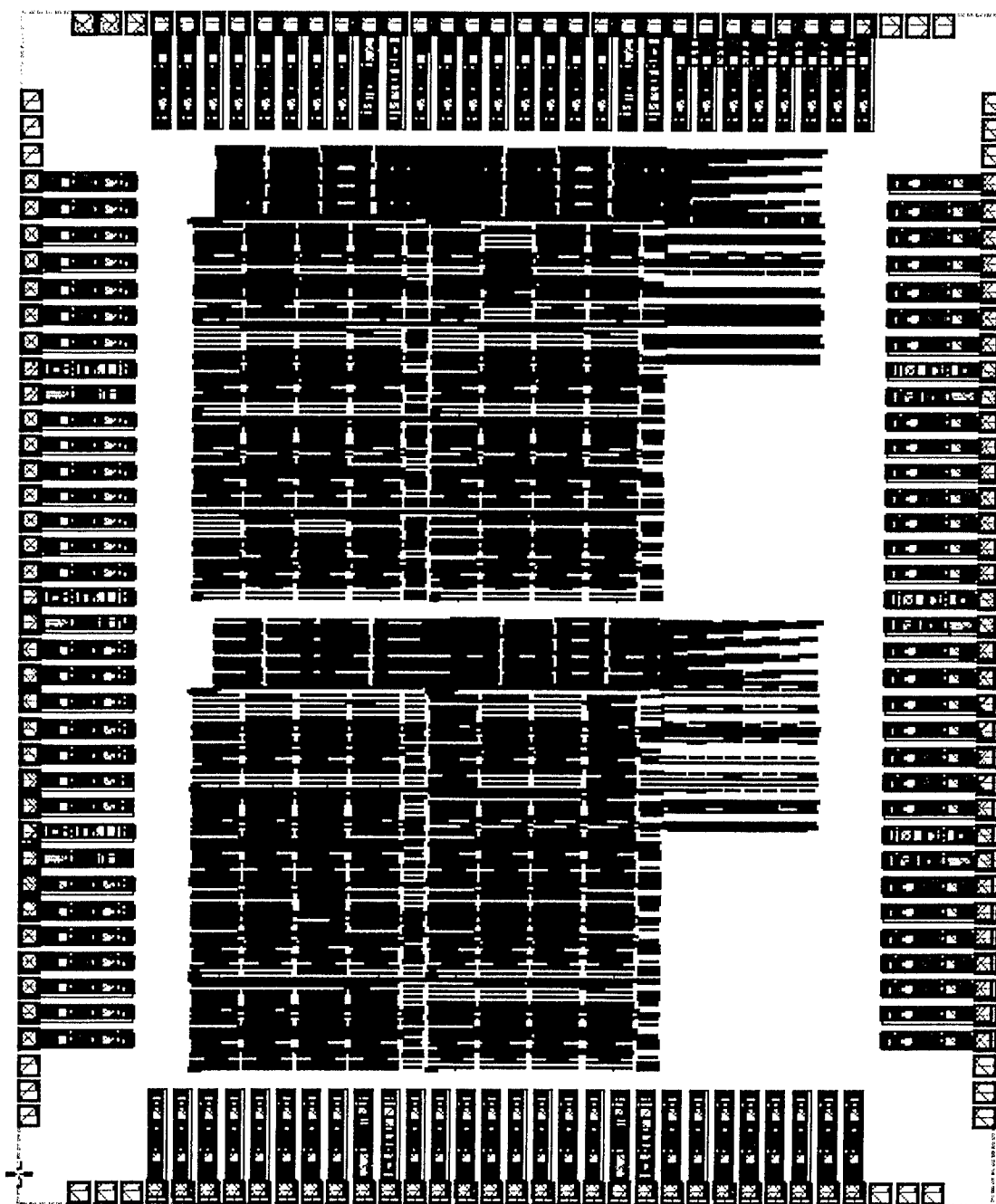


Figure 3.7.6: CSRC IC Layout

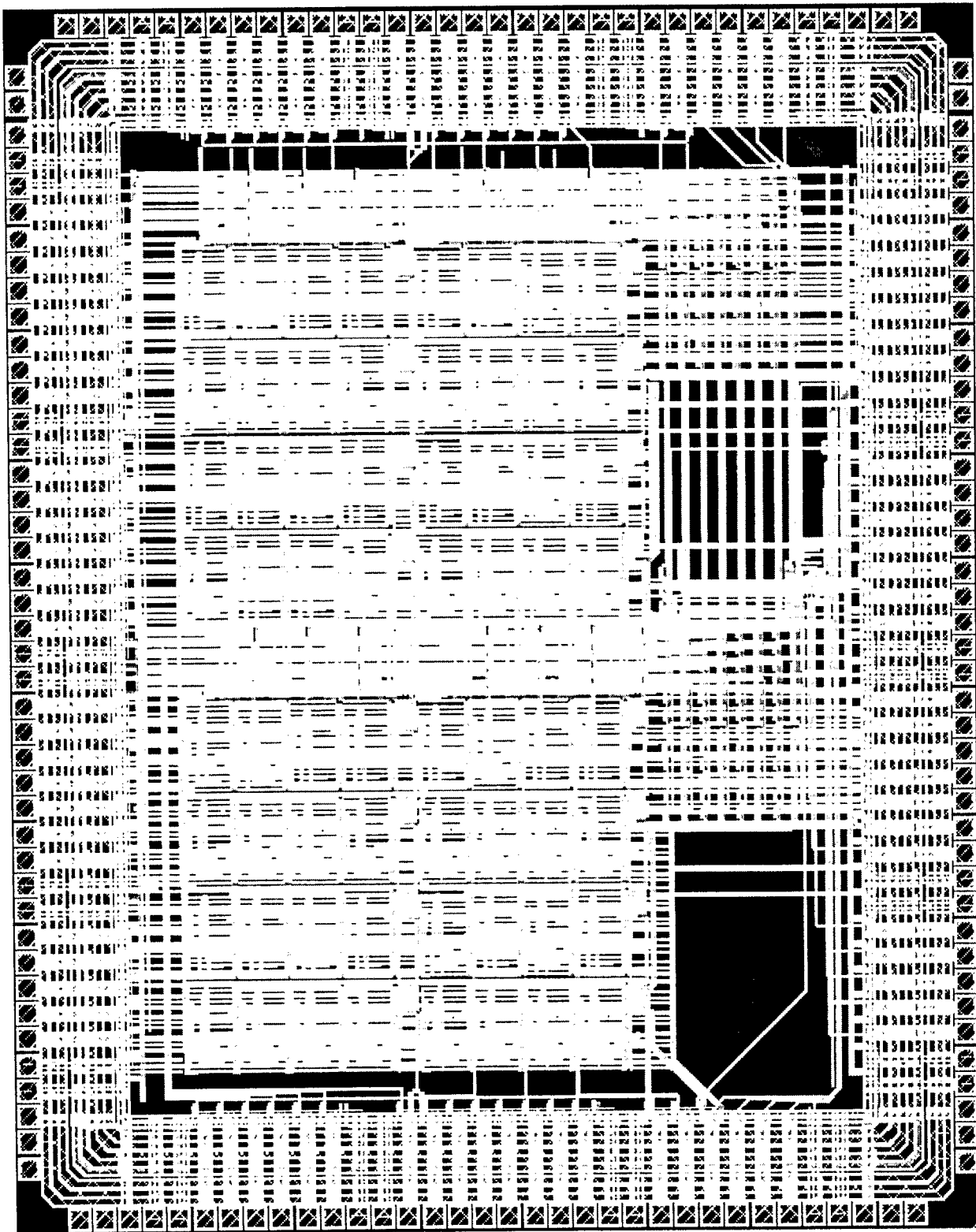


Figure 3.7.7: CSRC Prototype IC Final Layout

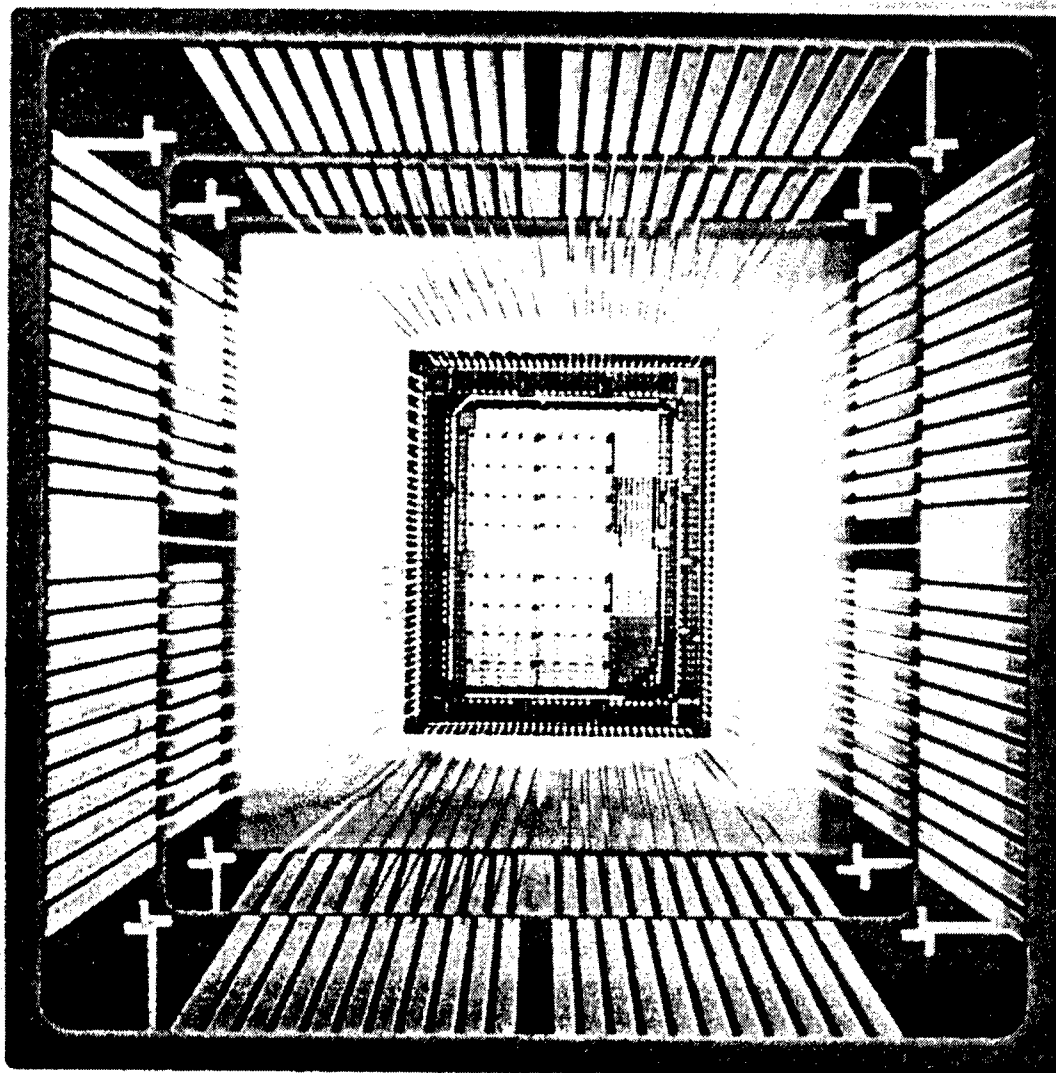


Figure 3.7.8: Photograph of CSRC Prototype Chip

3.8 Prototype IC Debug

The process of debugging the CSRC devices (both prototype and final) involved the use of new tools and methods developed by Sanders. The first run success of the CSRC prototype and final CSRC die serves as a testimony to the accomplishments of this effort.

Device model - The first step toward the verification of the design was to create a model that could be used to study the behavior of the design. To create this model, the original schematics were used to create a VHDL model of the entire device. This approach has both some advantages and disadvantages.

The first apparent advantage of using the schematics as a base for the VHDL model was simple expediency. With the schematics in place a simple exportation routine automatically generates colossal amounts of VHDL code that form the bulk of the model. With this code in place, it is then structured into logical libraries that form the model. It is still necessary to hand craft VHDL models for all the low level primitives in the design as well as design testbenches to verify the correct functionality at various stages in the hierarchy. The second, and perhaps most important advantage is the fact that the created model reflect every structural nuance of the actual design. Therefore by verifying the model, we are in fact verifying the functionality of the schematic itself, which in turn is compared to the actual layout of the device. This proves to be a powerful methodology that can virtually eliminate functional errors in the design. The first run success of the prototype device is clear proof of this. Oddly enough, the main disadvantage, also stem from the close relationship between the model and the structural design. The resulting model is so detailed that it imposes an enormous tax of the computing resources. None the less, the advantages far outweigh the disadvantages.

Another important factor of this structural model is that it exactly mimics the behavior of the device when it comes to programming. The clear advantage here is that it verifies both the validity of the bitstream as it will be passed to the actual device and the proper design of the programming chain (which encompasses a rather large percentage of the overall device). This feature does not come without a price however, as it places a heavy burden on the designer to produce a valid bitstream. Since the proprietary tools for the device were developed concurrently with the device, they were not available to the designer. This situation, coupled to the fact even the bitstream for the relatively small prototype device consisted of more than four thousand bits, meant that programming the device would be an arduous task. For this very reason, a series of Perl scripts were developed with the objective of facilitating bitstream generation.

Bitstream generator - The Perl based bitstream generator developed is called pport. It defines a simple configuration description language specifically design to target the CSRC device. Pport works in a similar fashion to a language compiler. As a command line tool it will take as arguments the name of the target project and a list of files containing the description of the configuration for the device. After execution, the resulting configuration is translated into a bitstream that can be downloaded into a CSRC device. The bitstream can optionally be stored as a VHDL file that can be used to drive

the model for the entire device. Simple modifications of the basic scripts have been made so that the CSRC Toolkit can leverage pport to save the bitstream in binary.

The pport program greatly facilitates the job of the designer by giving every bit in the configuration a tag that is easily understandable. For example:

```
BYPASSFF 1
```

Means that the flip-flop inside the Logic Cell will be bypassed. At first glance, it might seem that this approach is not terribly intuitive. The natural alternative to this method would have been for example to find the one thousand six hundred and thirty second bit in a stream of one's and zero's and changed it by hand. Note that the previous example is not complete. In a real world scenario you would want to specify what logic cell or flip-flop you are referring to. This is accomplished with a set of simple addressing commands.

```
CONTEXT      2
ROW           3
COL           5
LC            12
BYPASSFF      0
```

This example uses the flip-flop (this time BYPASSFF has been set to 0, thus enabling the flip-flop) in the twelfth logic cell in the logic array in the third row fifth column. This setting takes effect only when the device is switched into context two. Similar tags are defined for every possible configurable feature of the device. Although these tags alone make the bitstream more readable it would still be extremely tedious to enter an entire configuration by this method alone. Any task that involves so much work and is entered by hand is bound to be error prone.

The first improvement that pport provides is a set of default values. Any tag that is left unspecified by the user, will automatically be assigned a default value. The pport suite includes it's own set of configuration files that not only defines all the valid tags, but all determine what the default values should be. In addition a set of files can override default values for any tag. For example:

```
default
BYPASSFF      0
USE_CARRY     1
SUBTRACT      1
```

This sample essentially turns every user-unspecified flip-flop active. It also activates all carry logic circuitry and sets it up for subtraction. The user can always override the configuration for a specific instance by simply targeting that instance's tag directly.

Another time saving feature is the concept of a common configuration. The user can set a group of tags to their desired values, but instead of targeting an instance directly the tags are set for a common configuration. This common configuration can then be used to quickly define several instances at once.

```

common
1111 1
1110 1
1101 0
1100 0
1011 0
1010 0
1001 1
1000 1
0111 0
0110 0
0101 1
0100 1
0011 1
0010 1
0001 0
0000 0

ctxt 0
col 2
row 2

fillcslc 0
fillcslc 1
fillcslc 2
fillcslc 3
fillcslc 4
fillcslc 5

```

This example demonstrates how to define a common configuration for a look-up table. It sets its behavior to an adder configuration. After that a series of fillcslc copies that behavior into six specific logic cell instances. In essence this sample defines a six-bit adder. The section that defines the common configuration could be set in a separate file and could then be used as a relatively placed macro. This capability enormously increases the productivity of the user.

In a similar manner, a set of dedicated commands facilitates the setup of bus routes. A set of simple commands allows rapid definition of routing matrices for bus oriented routing. A single command can automatically set the values of as many as 64 bits at once.

Once all the bits in a configuration are set, pport allows the user to view the results in a number of different reports. These reports allow the user to quickly asses the configuration of the device from a high level perspective. Finally the results can be stored as a VHDL file which will drive the simulation of the CSRC model.

Windows NT based CSRC Environment - The pport bitstream generator was originally developed with a command line interface under a Unix platform. This arrangement worked well during development and testing since the CSRC was also designed in a Unix environment. A migration to a Windows NT environment was prompted by the development of the CTB board. The CTB utilizes a PCI interface. This is a clear

advantage due to this interface's enormous popularity. It facilitates the distribution of the test board to any interested party. Nonetheless, this imposes the creation of a Windows NT environment for the CSRC tools.

Fortunately the tools designed by FPGA Technologies work under Windows NT, therefore no porting was necessary. In the case of the pport tool, the port was seamless. This is a result of PERL's cross-platform capabilities. The only changes involved were the redefinition of environment variables pointing to the scripts' home directory and several custom files within.

In addition to porting pport over to Windows NT, it was also integrated into an inexpensive yet powerful and flexible text editor. The Editor is called Edit Plus and it was developed by ES-Computing. This editor allowed several features that facilitate configuration development. One such feature is syntax highlighting. By simply providing a syntax definition file, the editor will highlight all keywords recognized by pport.

Another interesting feature afforded by the editor consists of an integrated template feature. The editor will display a panel on the left hand side that contains all the possible tags available in the definition of a bitstream. When the user selects a tag from this panel, a placeholder will be inserted in the code. The placeholder will include the tag itself, a sample value for the tag, and a commented description of the purpose of the tag. This serves as a quick reference for the user.

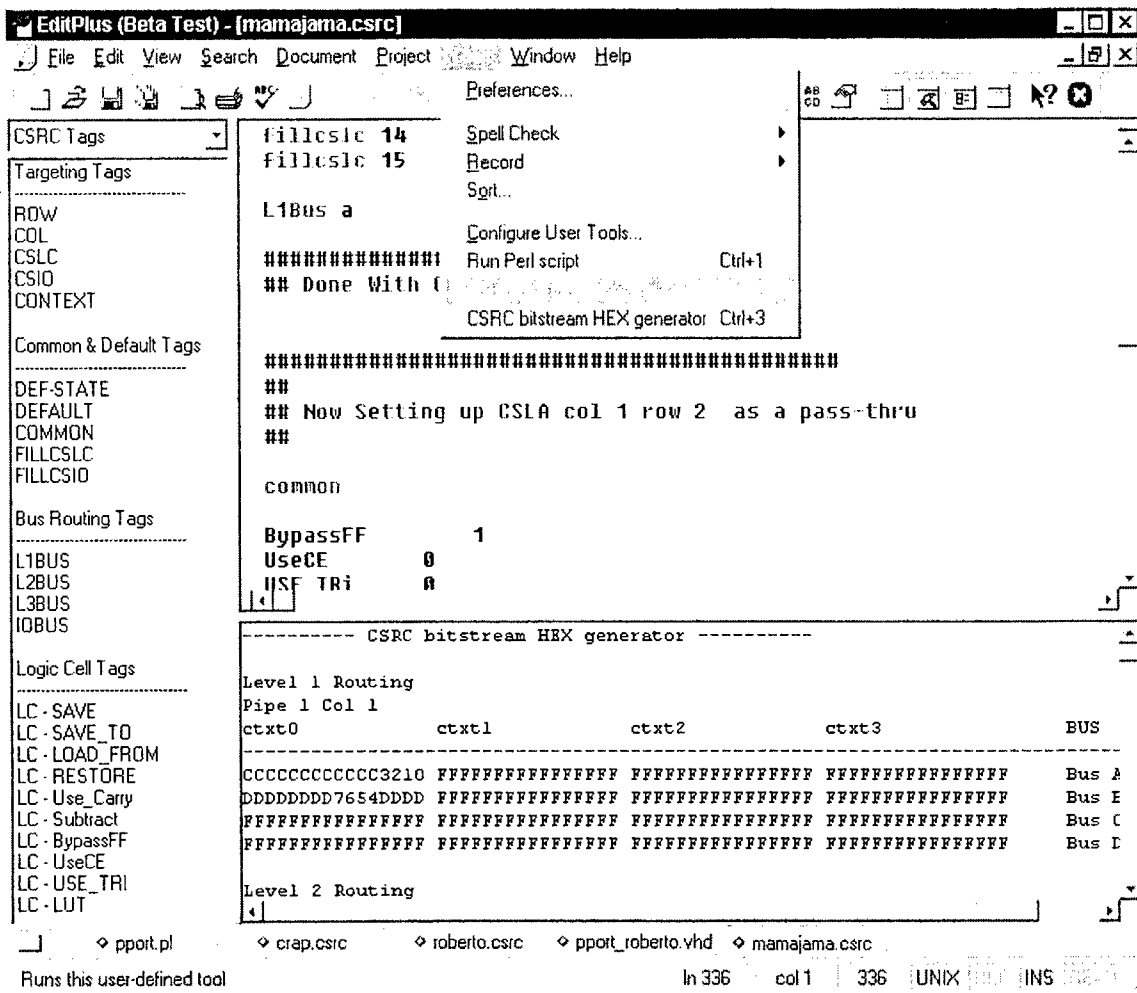


Figure 3.8.1: CSRC Prototype Programming Tools

The final feature is the integration of the pport program itself. With a simple keystroke, the pport program gets executed on the file being edited. When the program has executes, its output gets captured into a panel. If pport finds errors in the file, it will print out a list of the offending lines. By simply clicking on the error on the capture panel, the corresponding line in the code will be highlighted in the main editing panel.

Figure 3.8.1 above illustrates Edit Plus and its tight integration with the pport program. On the left side of the window, the template panel can be observed containing the various CSRC tags available. The output panel containing reports from pport can be seen near the bottom of the window.

3.9 Concept Validation Prototype (CVP)

The original purpose of the CVP was to have a board that could mimic context switching by utilizing commercial FPGAs and tying all of their IO together. In this manner, by having a control line that tri-stated the IO from one of the FPGAs (while the IO of the other FPGA was active) it was possible to switch between FPGA outputs on a clock cycle. Unfortunately, this idea would not have been able to mimic any of the data sharing capabilities afforded by context switching. However, the CVP would provide a platform for experimenting with dynamic reconfiguration.

During February 1997, Sanders finalized a list of required and desired functionality for the Concept Validation Prototype (CVP) module. Required features include a control chip, PCI bus I/F and associated control logic, DRAM management, on-board memory, and two devices with all pins tied together except those pins required for configuration (ping-pong). Desired features include connector to extend the SIMM bus, A/D converter, a microprocessor socket, and a possible daughter card connector. Plans for the CVP were to base the module on the Xilinx 4036EX devices, and if a crossbar is implemented, using a smaller Xilinx or an I-Cube device. Note that at this point in the program, the baseline was still to have Xilinx develop a XC4000 based context switching device. The module was designed to be a standard 32-bit PCI form factor board with an overall estimated clock speed for the module of 33 MHz.

Sanders completed the electrical design of the Concept Validation Prototype module in May 1997, however was instructed by the customer to not fabricate the CVP module since it was believed that the board would not afford enough of the functionality of the CSRC devices to justify a board spin.

3.10 CSRC Test/Demo Board (CTB)

The CSRC Test Board (CTB) was developed to allow experimentation with CSRC devices, preliminary mapping of algorithms on a context switchable architecture, and general testing of the CSRC prototype ICs. The CTB was designed with the idea that the board would provide the user a simple, easy to use interface to the CSRC device.

A PCI bus interface was selected as the main interface to the board since this type of bus provides an industry standard, moderate performance bus architecture that offers low cost and allows differentiation. It is also amongst the most widely used in universities and industry where experimentation with this board would most likely occur.

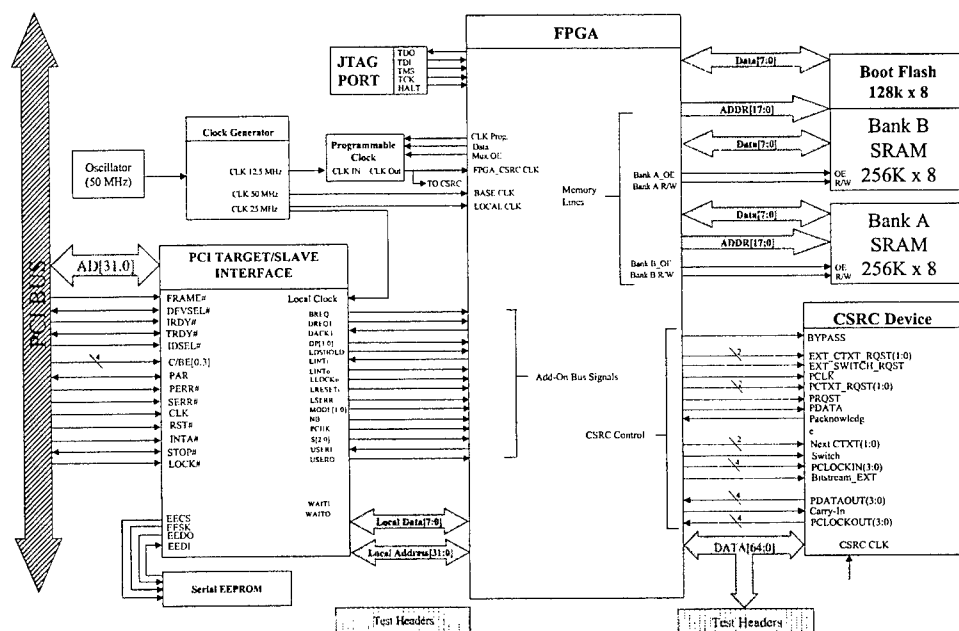


Figure 3.10.1: CTB Block Diagram

Access to the CTB from the PC is done through a PCI bridge chip that translates PCI bus signaling to a standard 80960 processor-signaling environment. The PCI 9060SD was selected as the PCI bridge chip for the board since it provided a single chip solution that minimized board space requirements and ensured PCI hardware compatibility compliance. The PCI 9060SD acts only as a target on the system acting purely as a slave to a PCI master. A Xilinx 4036XL FPGA drives the local side of the 9060SD chip acting as the local Memory and I/O controller. The CSRC prototype chip did not provide enough resources to implement both the test algorithms and the local bus arbitration needed. Therefore it was decided that an additional Xilinx chip was required to support bus arbitration and to handle miscellaneous and unforeseen functionality. The Xilinx XC4036 FPGA was selected due to its high I/O count which would allow probing of all of the CSRC I/O, while possessing sufficient capacity to support the arbitration signaling required to drive the 9060SD chip and miscellaneous resources such as the local programmable clock generator. The Xilinx FPGA boots from an on board Flash that contains CSRC configurations as well as the Xilinx configuration bitstream.

The Xilinx FPGA responds to accesses from the local bus by decoding the upper bits of the address. Possible destinations for the incoming data include the CSRC IC, the memory, and the programmable clock. Data can also be interpreted as an instruction that requires the Xilinx to pass multiple sets of data through the CSRC or reconfigure the CSRC. Memory was added to the board to allow for temporary storage of results required by some algorithms. The memory was split into two separate banks of 256k x 8 each to give the board some flexibility. Dual memory banks allow data to be read and processed from one bank while the results are stored on the other bank.

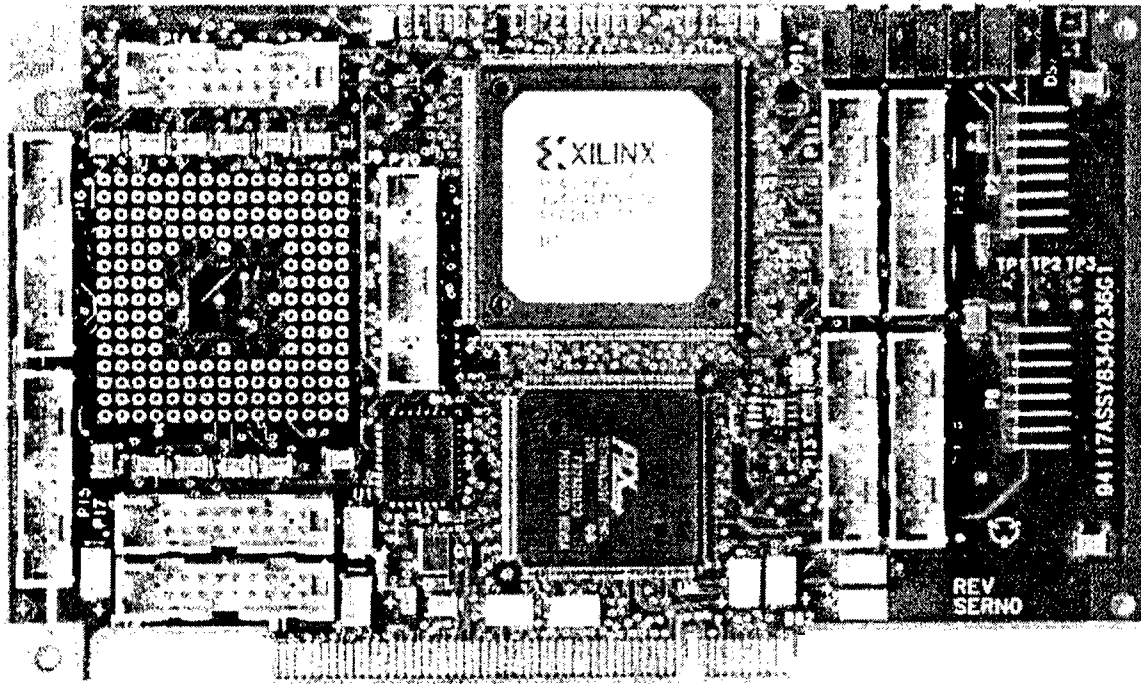


Figure 3.10.2: CTB Board

Since the CTB was designed as a testbed for the prototype CSRC device it contains a wide variety of debugging facilities. Every pin of the CSRC device is available for probing via test connectors. These connectors allow a digital oscilloscope to easily interface allowing the design to view all the signals within. Additional test connectors bring out signals pertaining to the local bus between the PCI 9060SD and the Xilinx 4036XL. A couple of spare pins on the Xilinx device were brought out to both test connectors and to a row of LEDs. These LEDs allows the user to program special flags or warnings into the Xilinx device and have an easy access to the current status. Finally, the board contains a flexible programmable clock generator that allows the user to select the frequency for operation.

Board Support Software

Both pport and the tools developed by FPGA technologies were anticipated to be able to generate bitstreams. The next piece that is needed is a software tool that is capable of communicating with the actual hardware and downloading the configuration. In the case of the CTB, that software came in the form of ProtoTester. ProtoTester consists of a

simple GUI that allows the user to communicate with the board in a PCI slot. Refer to Figure 3.10.3. To achieve this goal, proper PCI drivers were needed. Instead of spending time and money in the development of such drivers, a commercial solution was used. WinRT is device driver that allows software to make simple calls to the running driver to communicate with the actual hardware. The tradeoff was in execution time. WinRT involves some overhead that prohibits the interface to run at top speed. In our case, development time was deemed more important (Only in the CTB's case) so the driver was used.

The GUI portion of ProtoTester has several fields that allow the user to read and write data to and from the PCI board in a variety of ways. In addition to simple probing of memory, ProtoTester allows to download a configuration of the Xilinx part into the Flash RAM.

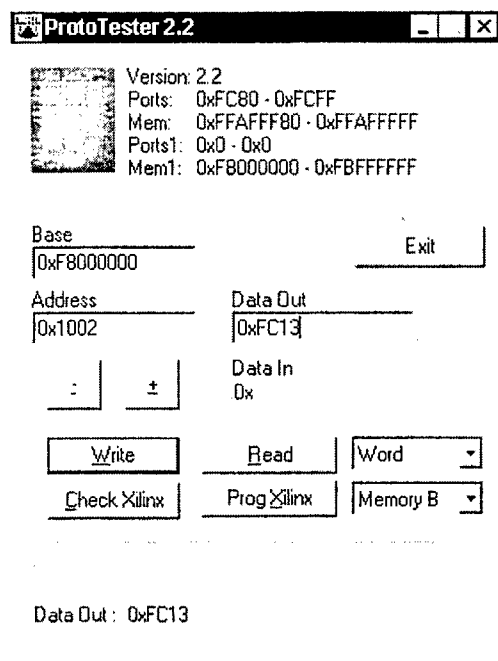


Figure 3.10.3: ProtoTester Screen Capture

The goal of ProtoTester was to allow users to download and manipulate the CSRC device from this program. It is capable of configuring prototype devices, switching contexts and running data through the prototype CSRC.

3.11 VHDL Model/ Schematic Capture for Device Simulation and Design Verification

Basic Context Switching FF Model

In the first few months of the program, a “manual” flip-flop switching & storage model that is context switching “aware” was developed and tested. The VHDL model is comprised of two fundamental components. The first part remains constant from design to design and involves code for the flip-flop itself as well as some related constants. The second part involves those aspects that change from design to design. All flip-flops are connected to a “main” flip-flop bank that actually switches the inputs of the flip-flop according to context switches. This main bank also provides global signals so that different context designs can communicate with the flip-flops. Also during this time period, an “automatic” flip-flop switching & storage model that uses PERL scripts to automatically generate the VHDL model for the flip-flop bank was also created and tested. The PERL script takes in files that describe the flip-flop requirements and configurations from different design contexts. The resulting VHDL model is configured to behaviorally mimic the configuration and routing of the multiple contexts of the design. With this mode, it is relatively simple to create hand-written behavioral models that use the flip-flop bank, thus enabling the early exploration of context switching issues.

During December 1996, the initial context-switching VHDL flip-flop model was developed. This model was tested with a handwritten VHDL testbench. Once the model was tested, a Perl script was written to automatically generate a bank of context-switching flip-flops based on a flip-flop requirement file(s). To describe the flip-flop requirements, a simple language was developed, this language allows the user to indicate how many flip-flops are needed, under which contexts they should operate, and how they should be configured to operate at the context switch instant. Once this Perl script (called flopper) was operational, it generated several VHDL files required to make context switch possible. It was later expanded to include some amenities for manual programming. For example, a number of flip-flops can be grouped together to be treated as an n-bit register, each bit having identical context-switching behavior. The Perl scripts will then generate VHDL templates so that engineers can then hand write a behavioral context switching model. The tools thus far developed, were tested by creating a simple context-switching behavioral FIR filter model. This model used several features of the flopper script and used the flip-flop under several different circumstances (sharing data among contexts, one contexts write, others read, etc.) This test model development extended into January.

The next steps in the development were shared pin switching (similar approach to flip-flops), extraction of flip-flop requirements from LCA files (FPGA design files), and automation of that process via Perl scripts. Once these goals were completed Sanders had successfully developed a set of tools, with which possibilities of context-switching could be explored. This model, as it stood, was used to investigate the commercial viability as well as to prove the utility of the computational model of the original architecture and several of its envisioned features.

Structural Model of CSRC Device

In an attempt to maximize the likelihood of success on the development of both the prototype and the final CSRC IC, a structural model was created for each device. VHDL models for *all* of the low level components such as flip-flops, muxs, and gates were created and functionally tested. These models were then imported into Mentor Graphics and associated with symbols. With these symbols larger higher-level components were created within Mentor Graphic's Design Architect. Since these components were created with our own VHDL-based symbols, they too could be exported to VHDL and would properly link with the lower-level components. This procedure allows for faster graphical design with all the advantages of VHDL simulation. By directly associating the VHDL model with each low level component it was insured that a functional simulation of the device meant that the schematics were functionally correct. In addition, once the "layout-vs-schematic" LVS tool was run to ensure that the layout matched the schematic, we were guaranteed that the layout now was "functionally" correct. This simulation coupled with SPICE simulations of the low level components verified both functionality and timing.

As the model grew in size, the bitstream, required to program the behavior of the components in the VHDL model, became cumbersome and tedious to write by hand. For this reason, Perl scripts were developed that rapidly generate efficient VHDL code used to generate the correct bitstream to program the device. This Perl scripting effort will evolve as the CSRC model progressed and eventually served as a simple text-based design tool for the device.

Final CSRC Device Model

In an effort to strengthen the low-level component verification effort for the final IC, a new process was created that is centered around an internal website available to everyone in the development team. With this instituted system, individuals could post and verify progress on each and every component.

Although each of the functional building blocks was vigorously simulated, the control block was the element that received the most scrutiny since its failure would render the entire CSRC device inoperable. The control block, utilizing its VHDL model, was simulated under a number of different situations. It was first tested to do a bitstream download. The bitstream needed to be simulated before context switching can be tested, since in the new design the control block will only allow contexts that have been previously programmed to be switched into. Once a bitstream download had been simulated, a context switch was tested. Consequent tests involved staging conflicts that the control block might actually encounter during normal execution. It was then verified that that block would actually resolve such conflicts in the desired manner. For example, both a context switch and a bitstream download were requested at the same time and targeted the same context. The control block would then grant one of the request and deny the other (In this case the programming request was honored while the context switch request was declined). Further tests involved conflicts between internal and external requests for both switching and programming. A few minor bugs in the design were ironed out of the design after this lengthy testing procedure. Finally, the larger

block called CB_Unified, which includes control signal bypassing and security features, was tested. The Control Signal Bypassing feature allows the user to bypass the control signals generated by the control block itself thus allowing him to either debug the chip or to modify the interface protocol for downloading and programming to the device.

At the higher levels, simulation of functionality was rooted in simple CSRC design configurations. Once the Logic Array was targetable from the Perl scripts it was programmed as an adder to test the local routing, the look-up tables, the carry logic, registers, and various other programmable options. Next, variations on the simulations were run to test other circuits such as the dedicated routing between neighboring arrays and the shifters. When everything was tested and those few bugs that surfaced were repaired, all of the cs-bits were revised to test for BIST functionality. Essentially when the INIT signal is asserted, every cs-bit should control logic such that information flows from dedicated routing in through the array and out to the neighboring Array.

The CS_Pipe in the final CSRC device is a very large block on its own. It is approximately twice as large as the entire prototype device. Simulation was seen to be two to three hours for a single context. This large block includes, repeatable modules including the logic array, both level two and level three routing, two block memories and a routing matrix for internal communication to the control block. After analysis of the Pipe simulation and some extrapolation, it was determined that the final simulation would require approximately 2.8 Gigabytes of physical memory and would execute in about 12 hours. Such requirements far exceed the capabilities of the machines currently available in Sanders' Engineering Design Environment (EDE). Fortunately, the CSRC team was able to gain access to a Sun server with 6 parallel processors and 6 Gigabytes of physical RAM. As a result of Sanders investment in this workstation and the acquisition of a new LVS tool, Sanders was ultimately able to simulate the CSRC device at the highest level. Unfortunately, only one pipe was able to be simulated at a time due to the severe processing demands imposed. Hence, seven "dummy" pipes were inserted into the CSRC_CORE for simulation purposes. These dummy pipes were simply empty architectures and entities, therefore they did not add and significant simulation overhead.

3.12 Final Integrated Circuit (IC) Development

IC Modifications to the Prototype CSRC FPGA

Although the CSRC prototype had first pass success, it was merely a functional subset of the larger capacity final CSRC device. In addition to adding many additional features, the sheer capacity and die growth seen by the larger CSRC device demanded numerous IC design changes from the prototype. Albeit the prototype provided a strong foundation for some of the key building blocks. An architectural evaluation of the CSRC FPGA was conducted resulting in a finalized architecture for the next generation CSRC FPGA. Key features and capabilities that will be added to the CSRC prototype FPGA are as follows:

(1) An 8 bit RAM in the logic cells (2) Eight 256 x 8 dual-port synchronous block RAMs - 2 per pipe (3) In place logic array data shifting (4) The ability to download via the active context. It should be noted that a number of modifications and small additional features will additionally be incorporated in the next generation CSRC device, however, it is believed that the cited features are most significant and require the greatest design effort. The final IC design effort focused on three primary areas: (1) Designing a 256 by 8 bit on-chip block memory (2) Adding the capability of shifting the output of a logic array and (3) Redesigning / laying out of basic cells. Several high level components were rebuilt based on the prototype schematics. Many new changes in basic architectural infrastructure were adapted to each of these new blocks. Some examples of these changes are presented in the following list:

- Programming chain clock buses now include both the normal and inverted version of each signal. This requires more routing resources but dramatically reduces overall size of individual components that use these signal (CS_bits)
- Both standard and inverted versions of the user clock are being routed. Once again this is an area vs. routing complexity tradeoff.
- Programming bits no longer contain resets but rather drive a default value as long as the INIT signal is asserted. This greatly reduces the size of CS_bits (which in turn, are the dominating factor for the overall size for the entire device)
- Programming bits on every block are being revisited so that a BIST is in place during the assertion of the INIT signal. In addition CS_bits are being placed such that a bitstream formed of all zeroes in effect programs a "safe" configuration (thus making it easier for hardware or even software to turn off parts of the design by simply placing zeroes in the bitstream).

With all of these fundamental changes in mind, the Logic Cell was recreated conforming to the new standards. In addition the Logic cell now contains a small 8-bit "distributed" RAM.

As was seen in the development of the prototype, creating a single logic array requires the creation of four very similar yet distinct slices. Individual schematics and layouts were created for each. Each slice was then examined to determine differences between layout extracted netlists and the schematics. Once all the individual slices passed

examination, they were connected appropriately to form the larger array. At this level individual nibbles of information come together to form the final level 1 buses.

Several tradeoffs were considered for the level 3 routing. The use of bi-directional buffers created by opposing tri-state buffers controlled by the appropriate bitstream bits was considered. It was thought that the buffer entering and exiting level 3 buses would speed up signal propagation compared to the passive pass gates previously used. Unfortunately the collective gate capacitance of the tri-state buffer hanging on to level 2 signals considerably degraded level 2 performance, even in situations where level 3 buses were not being accessed. For that reason we reverted to the use of pass gates and are currently implementing a level 3 matrix based on them.

The L3 Matrix was divided into slices that were distributed along the entire pipe. This distribution results in a tighter, more efficient layout. To increase overall performance the Level 2 bus connections to Level 3 busses are no longer bi-directional. This results in better performance since a buffering tristate now replaces a pass-gate. The implication is that only certain L2 busses can drive onto Level 3 routing and only certain Level 2 busses can be in turn driven from Level 3 routing. There is actually a 2 to 1 ratio of Level 2 busses that can be driven from Level 3 to those that can drive onto Level 3 routing. This change alone might diminish overall Level 2 flexibility. However, due to layout symmetry it was actually possible to double the number of Level 2 busses. There are now twelve busses per pipe, up from the previous 6. This increase balances Level 2 flexibility with regard to Level 3. In addition, this change actually dramatically improved routability within a pipe without dramatically increasing the required physical area.

The new IO cells were updated to utilize National's new 4 layer metal cell. In addition, the connectivity between Level 2 routing and the IO cells was revamped to include a 4-input mux on each IO cell. This feature affords the placement tools the added flexibility for combinatorial logic.

The new control block design boasts several new features including a BIST mode, parallel programming, and security features. An offshoot of the wider bitstream is a wider preamble sequence, which further reduces the chance of random bits on the lines; triggering an accidental programming sequence after a program request has been made. In addition, Internal programming was enabled. This means that the current context is now not capable of requesting a context switch but also to reprogram an inactive context. To allow more flexibility a new "ghost" Level 3 switch matrix has been designed that affords the user to route reprogramming and context switching signal from anywhere within the device. To provide more security features a new security block has been created to switch off key external pins. This block has it's own external pin. When the pin is pulled up by the user once, the block will shut off sensitive pins until the next power-off.

IC Physical Layout Size Reductions from the Prototype CSRC FPGA

Since the next generation CSRC device contains 16 times as many logic cells as the prototype CSRC device and is comprised of over 8.5 million transistors it was incumbent

upon the IC design effort to ensure that the core elements of the full-custom design are optimized and physically compressed. For this reason, the cells that are repeated throughout the IC design, and therefore drive the physical area of the FPGA design, have been redesigned with an average reduction in area of about 30%. Additionally, the cell interconnections have been modified at all hierarchical levels to accommodate the new smaller core cells.

One of the first elements in the final device design was to revisit the blocks from the prototype chip in an attempt to reduce their physical area and make the core modules more robust if necessary. Since the die size was projected to be 650mils x 650 mils (actual size was 610mils x 610 mils), these two efforts were very important to the success of the CSRC final chip. In addition, any design refinements that can increase the quality of the FPGA design would obviously result in a better IC design at the conclusion of this program. With this in mind, design modifications to the programming D-flip-flop were made to avoid potential hold time violations and to reduce the physical size of the PDFF.

The new device has accomplished higher densities in part thanks to a new CS-bit scheme. Under this new scheme, Cs-bit flip-flops no longer have reset circuitry, thus reducing their size. Instead, the CS-bit uses the init signal as a select line that points to a safe BIST configuration. In order for this scheme to work the control block must ensure that the main CSRC core remains with its INIT signal asserted until a context has not only been properly programmed, but also switched into. In addition the control block also keeps track of which contexts have been properly programmed and which have not. This allows it to permit only switch requests to valid programmed contexts. Not only was the design of the CS-Bit refined, but the layout was compressed and made more regular for ease of tiling. See Figure 3.12.1 for a picture of the relative size layout of the prototype and final CSRC CS-Bits. Note that minimizing the physical size of the CS-bit has a tremendous impact on the final CSRC device as there are over 80 thousand CS-bits in the CSRC IC.

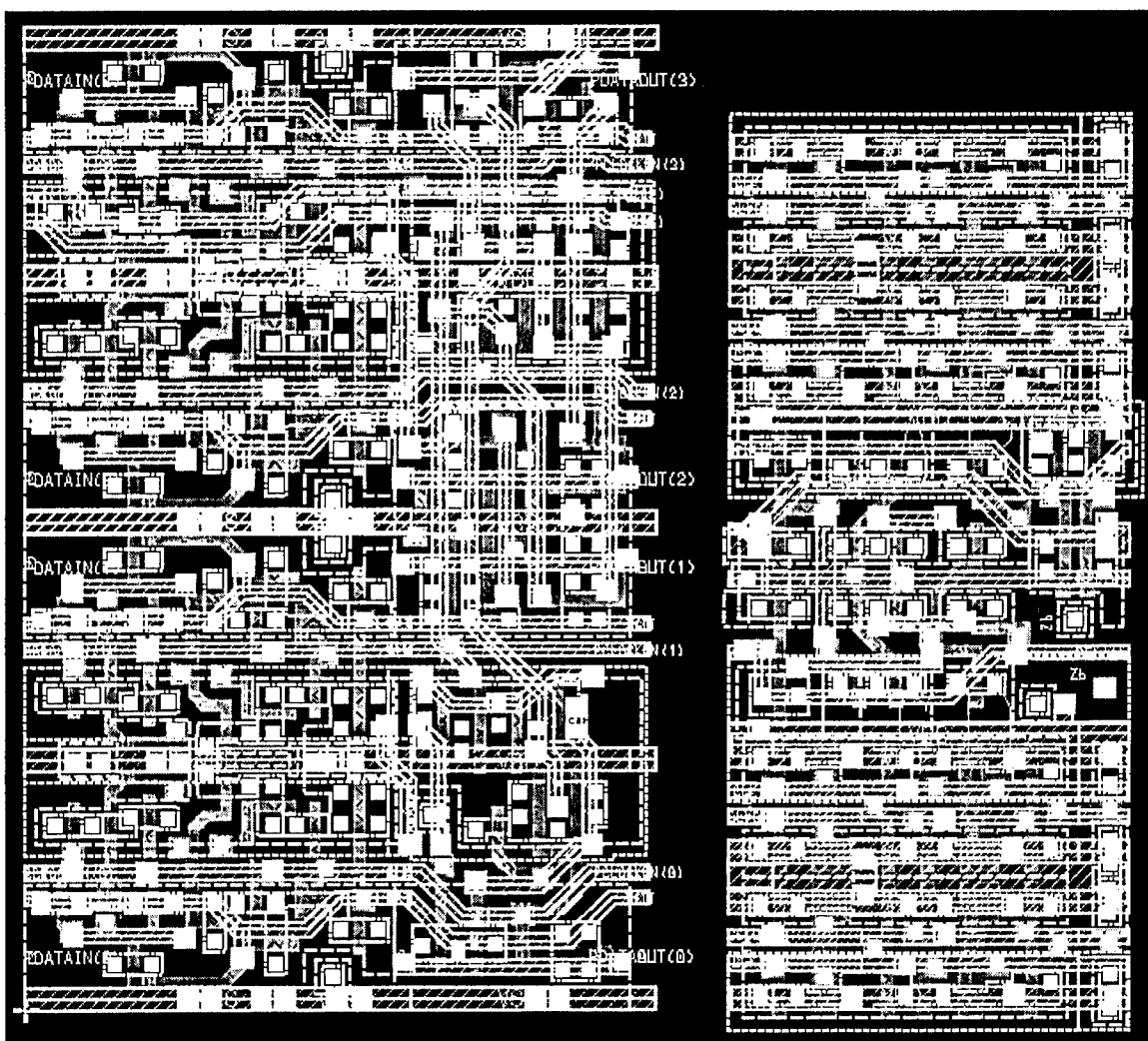


Figure 3.12.1: Relative Size of Prototype (left) and Final CSRC (right) CS-Bits

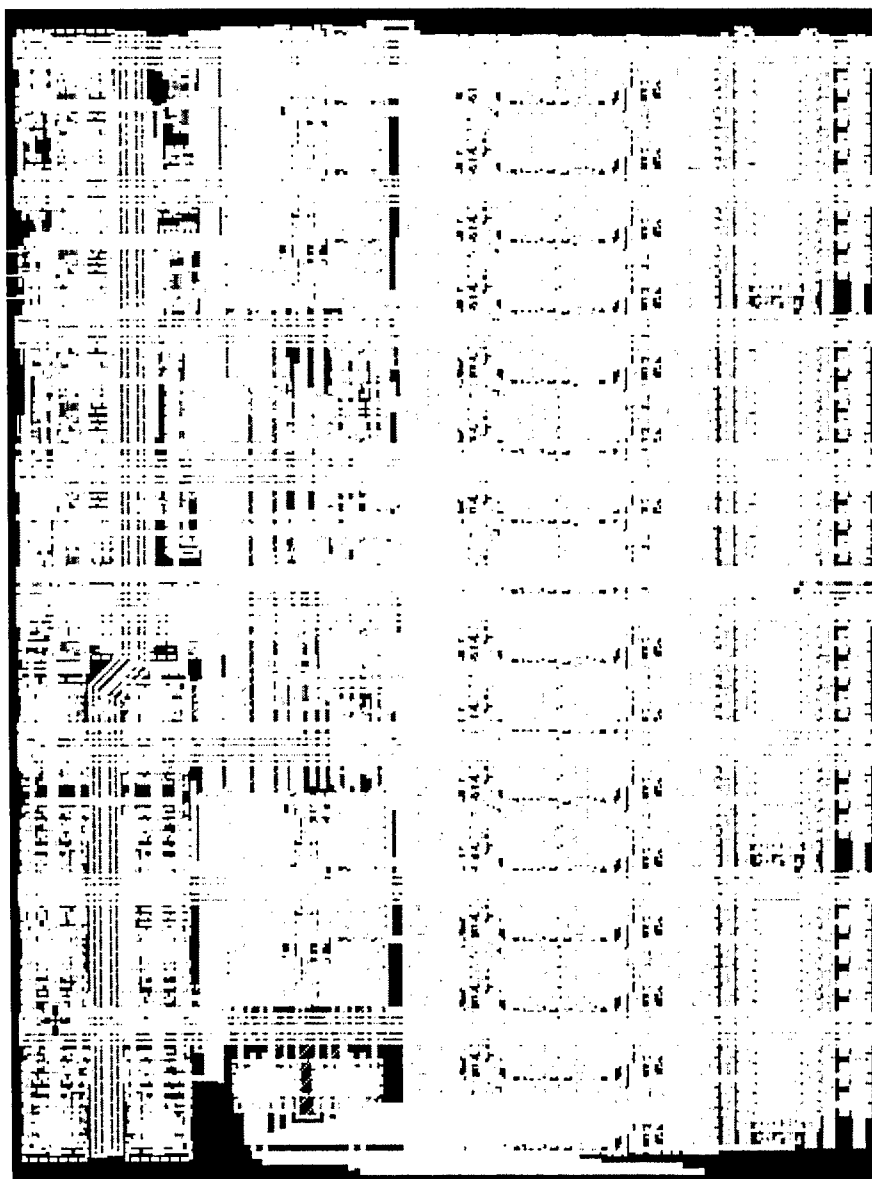


Figure 3.12.2: Layout of CSLC in Final CSRC IC

Block RAM

As a result of the architectural specification work, the final CSRC IC design was deemed to have 16 “block RAMs” distributed throughout the FPGA. Each of the sixteen on-chip block memory elements is a 256 x 8 dual-port synchronous static RAM. The 256 byte SRAMs afford the capability to perform simultaneous read and write operations. The schematic for a single bit of the RAM is shown in Figure 3.12.3 and the layout for a byte of the RAM is shown in Figure 3.12.5.

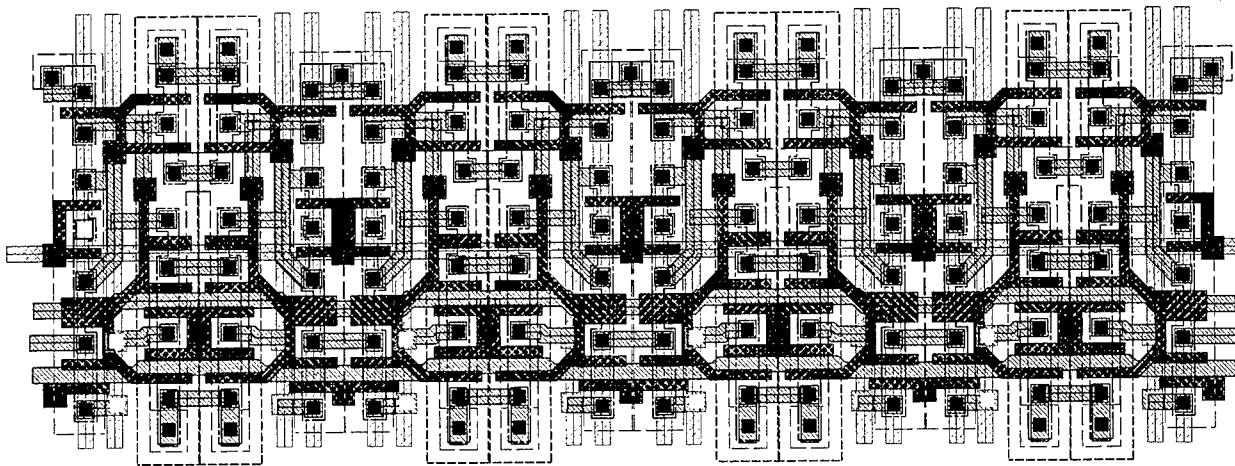


Figure 3.12.5: Layout for a Byte of Dual Port Synchronous Memory

The memory block has a read address, a readout port, a write address, a data port, and a write signal. Look-ahead circuitry was designed to detect identical read and write addresses. To accommodate this “pass-thru” case, a mechanism has been designed to allow readout directly from the data register. The basic dual port static memory cell consists of a six transistor memory cell together with a tri-state readout gate (see Figure 3.12.3). The inputs to the individual cell are: read enable (REN) and read enable bar (RENB), the read port (RP), the write enable (WE), the write port (WP) and write port bar (WPB). This choice for the memory cell increases write speed and provides readout robustness by eliminating the need for an analog sense amplifier and their inherent sensitivity to electrical noise and temperature. Eight cells together with the two gates accepting the X and Y decoder outputs for read and write operations form the basic SRAM byte (see Figure 3.12.4). The 256 byte memory is organized as a 16 row by 16 columns array with associated Y decoder for the rows and X decoder for the columns. The decoders are depicted in Figures 3.12.7 and 3.12.8. In the write operation the X decoder accepts the write signal. The output of the decoders is high only when this write signal is enabled. The outputs of the X and Y decoders are registered and fed to the gate that enables the write enable signal. In the read operation the 16 x 16 array is divided into 4 subarrays of 8 rows by 8 columns each as seen in Figure 3.12.7. The output ports of the individual memory cells in each column are connected in parallel, forming 8 to 1 multiplexers; one for each bit of the output word. This organization is chosen to reduce the read time to a reasonable amount while keeping the size of the output tri-state inside each individual memory cell to minimum. The output of each column is selected by another 8 to 1 multiplexer which differentiates between the 8 columns of each subarray. This is done for each bit of the word and results in 8 such multiplexers. The output of

each of the four subarrays are then connected to a 4 to 1 multiplexer forming the output word.

Simulation results have shown excellent performance results of the above described block memory.

Address decode time	$T_{\text{decoder}} = 0.6\text{ns}$
Time to stable readout port	$T_{\text{read}} = 2\text{ ns}$
Time to stable write operation	$T_{\text{write}} = 0.3\text{ns}$

Note that all simulations were done under worst case conditions of voltage, temperature and transistor processing.

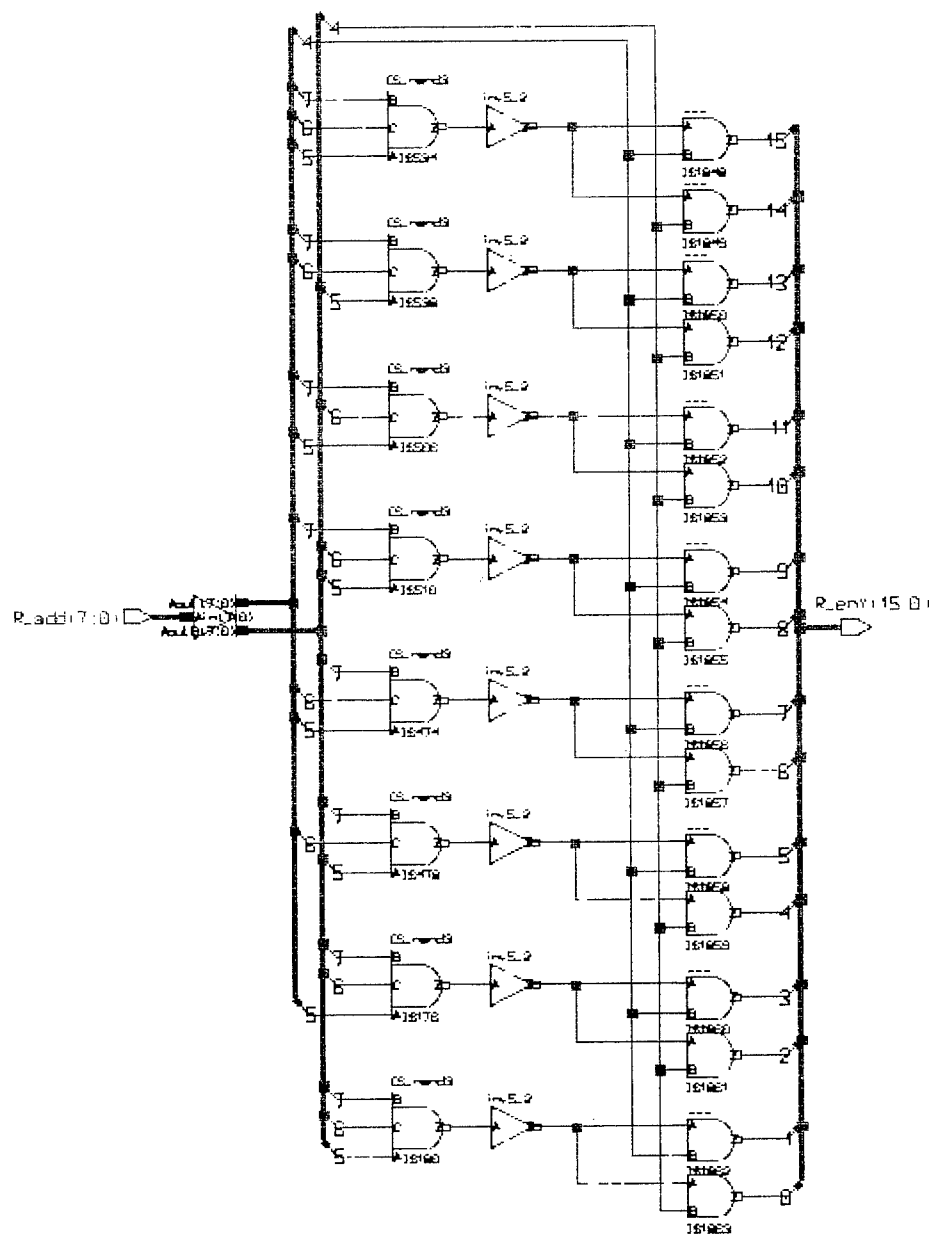


Figure 3.12.6: Context Switching Memory Decoder

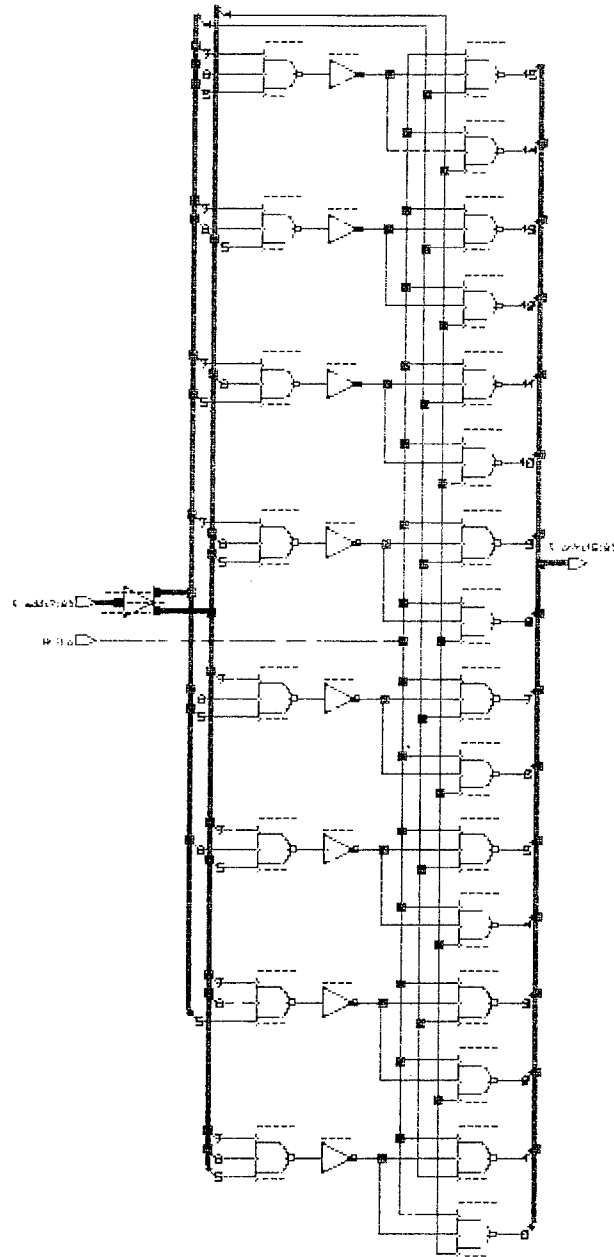


Figure 3.12.7 Context Switching Write Decoder

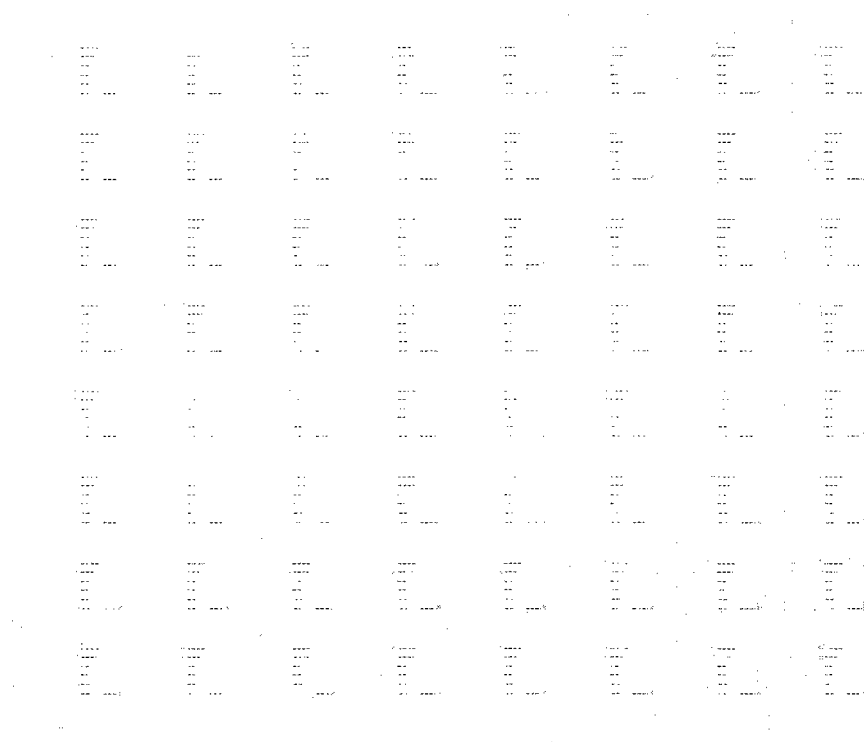


Figure 3.12.8: Context Switching Memory 64



Figure 3.12.9: Layout of 256x8 Dual-Port Synchronous Block RAM
(Including Interface to Level 2 Routing)

Logic Array Word Shifter

Shifting the output of logic arrays is useful in many logic and arithmetic operations. Since shifting can greatly reduce the necessary hardware when implementing constant coefficient multiplies, the architectural specification effort concluded that providing this capability would be a key enabling feature for efficient DSP algorithm implementations. The shifter in each logic array can shift the output word left by 0-7 bits. Each bit of the output word is passed through three cascaded 2 to 1 multiplexers shifting by 1, 2 and 4 positions respectively. Using a three bit code it is possible to control the three multiplexers and obtain any shift between 0 and 7. The implementation of the shifter is distributed between the 16 logic cells that comprise the logic array. The additional circuitry in each logic cell minimally consists of three 2 to 1 multiplexers and three buffers.

CSLC RAM (Logic RAM)

This RAM, which is included in each logic cell, consists of 8 enabled flip-flops modified for read/write operations. This design has been chosen for simplicity and for the speed of operation. The write operation corresponds to conventional data storing in the flip-flops. The synchronous read operation corresponds to disabling the input while reading the latest data stored in the flip-flop. After being designed, simulated, and aliased out, the 8 x 1 RAM results are identical to those for the enabled flip-flops and show delays between output and clock signals being less than 0.4ns. The incoming data (write operation) is as fast as a previously designed enabled flip-flop and of the order of 0.3ns.

CSLC

The logic cell designed for the final IC has added an 8-bit dual port synchronous RAM to the prototype IC functionality while simultaneously reducing the physical dimensions of the block from 150u x 454u to 135u x 280u. The Final CSRC CSLC is depicted in Figure 3.12.2.

Logic Array

The hierarchical design of larger blocks continued with the design of the interface to connect the logic cells in an array (16 logic cells) and 8 arrays in a pipe. Additional decoders were required for interfacing the pipes to level 3 routing. See Figure 3.12.10 for a depiction of the layout of the logic array. See Figures 3.12.11 & 3.12.12 for a depiction of the layout of a CSRC pipe and the clock tree, respectively.

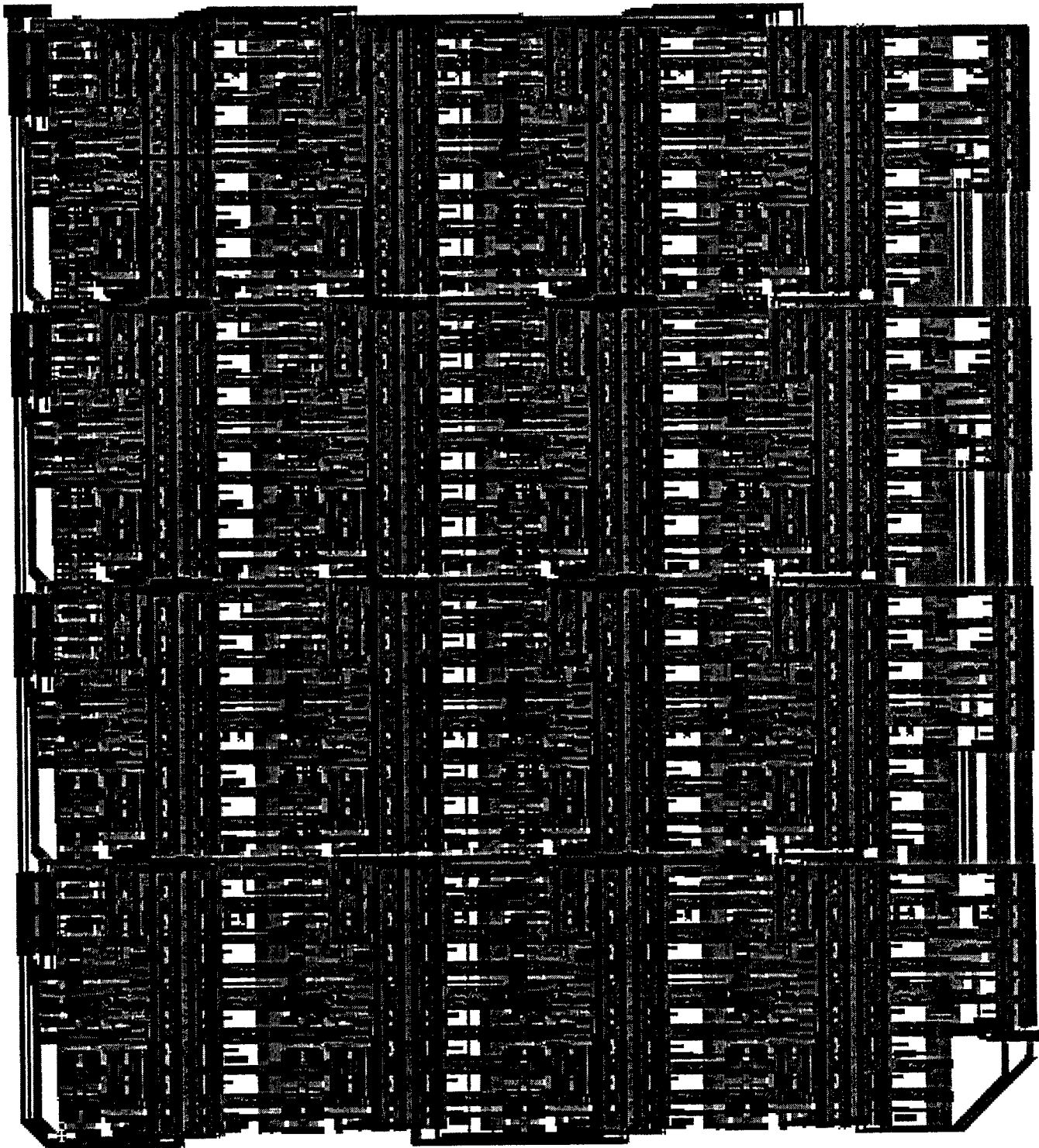


Figure 3.12.10: Layout of Logic Array (Including Level 2 Routing)

Overall IC Layout for Final CSRC IC

The individual *slices* that compose the context switching logic needed to be revamped since the prototype IC treated level 1 and level 2 buses as buses rather than individual bits. The snaking path of the configuration bits has been revised to come in and out of each slice allowing the snaking path to be completed in higher levels of hierarchy. The idea is to decrease the length of the overall path by making mirrored images of internal components. By using mirrored images, the traces that connect the path can be shorter since they do not need to travel around the perimeter of each individual component. See Figure 3.12.13 for a depiction of this new approach.

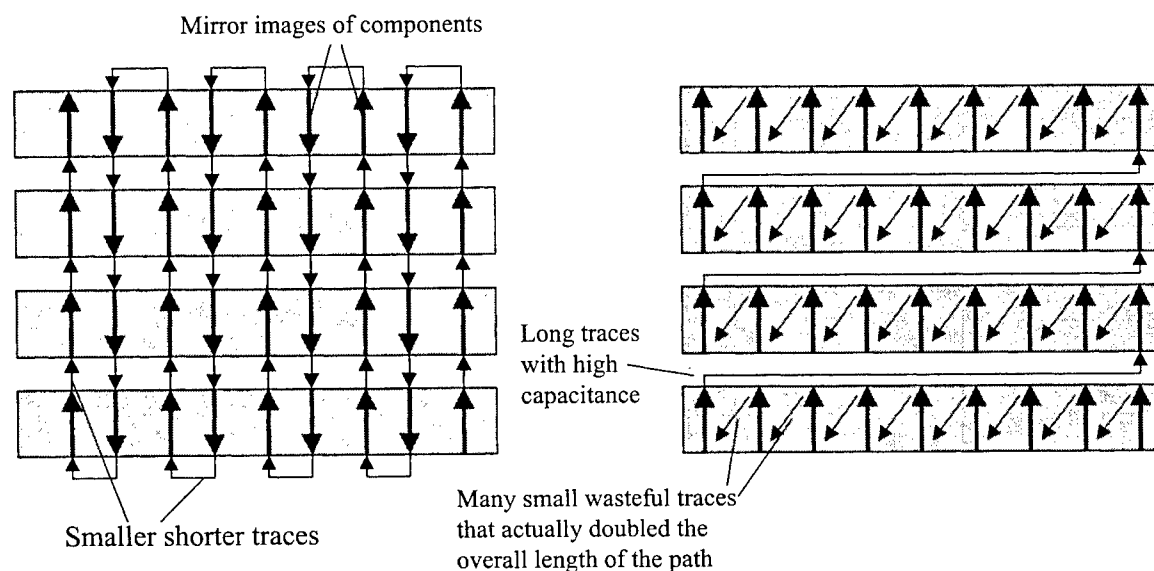


Figure 3.12.13 Context Switching Logic Array Minimized Bus Lengths

This modification required each component to be laid out in two directions. Although making a mirror image is not nearly as labor intensive as developing a completely new component, it was in fact not as simple as “flipping” the original. When a mirrored version is created, it affects the order of the bits in the bitstream, thereby reversing many bits from their original (prototype) order. This translates into separate versions of schematics for each, which in turn generate different versions of VHDL models, which in turn require better Perl scripts to program them. Despite the added effort, the improvements were considered essential in light of the considerably larger scale of the final device versus that of the original prototype.

Design Verification

Having completed the design of the Final CSRC FPGA (as seen in Figure 3.12.14), the process of testing the IC became the challenging task at hand. As was already described, with regard to design verification, the chip that was laid out went through numerous cycles of layout versus schematic (LVS) checking. The LVS checking guarantees that the schematics used for model simulation are an exact match of what was implemented using the layout tool. Further verification of the layout was done through the preliminary use of a second tool that checks the transistor level schematics using the model test vectors. This

secondary tool allows secondary verification of the physical implementation of the circuit on the chip versus the VHDL model.

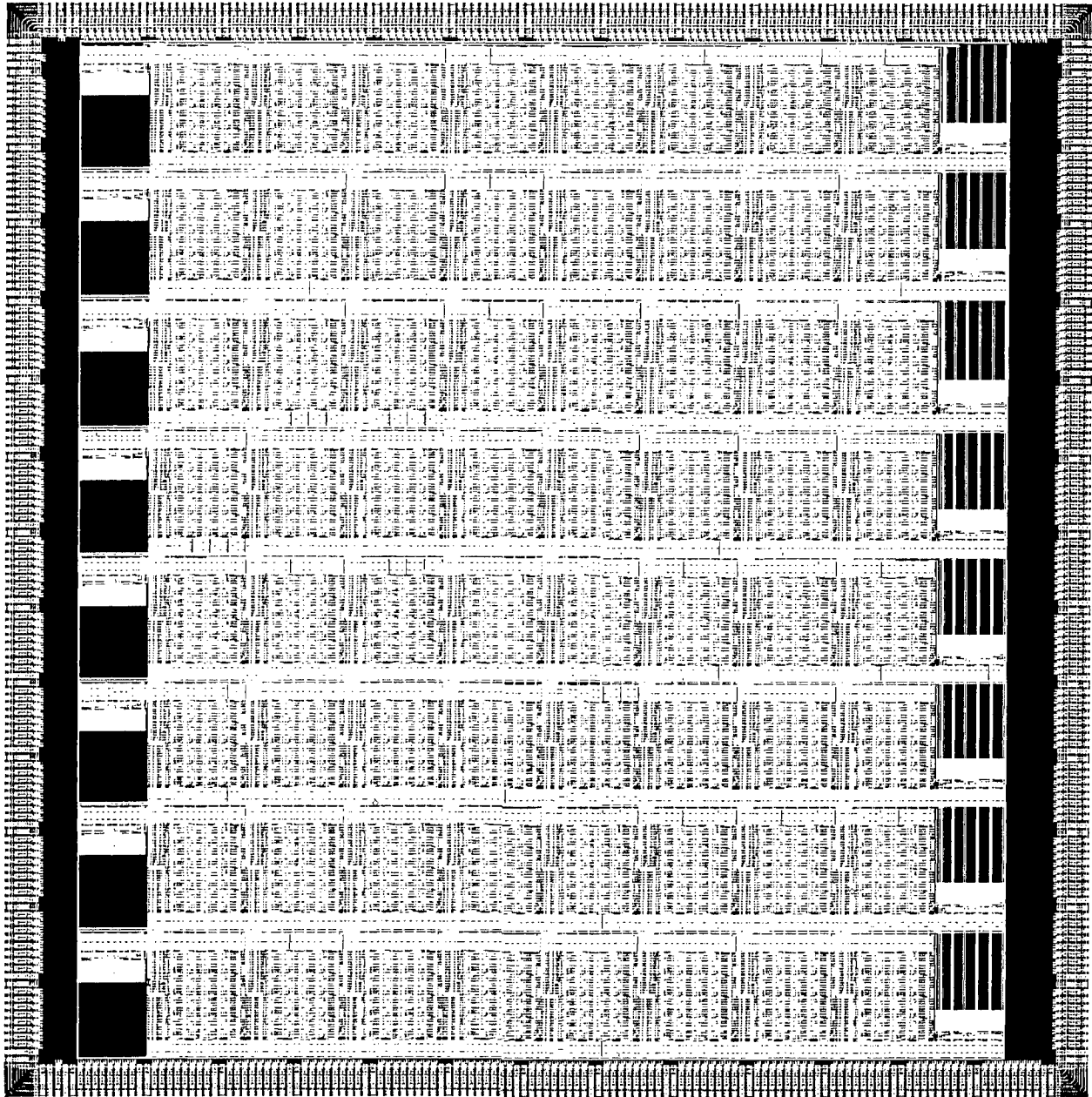


Figure 3.12.14: Layout of Final CSRC IC

Initial testing of the CSRC devices indicated that National Semiconductor had mounted the die 180 degrees off in orientation. This resulted in several power to ground shorts and rendered all of the devices unusable. National Semiconductor assumed full responsibility for this error and repackaged additional die at no cost.

With all of the test fixturing and test vectors are complete, the CSRC devices returned from the second fabrication and initial testing revealed that that die had been mounted

correctly this time and there are no power to ground shorts. In fact, testing indicated that we can successfully program all four contexts, switch contexts in a single clock cycle, share data, utilize the direct/shift routing, and perform the programmed logic function in the active context. Unfortunately, the tremendous flexibility of FPGAs makes exhaustive testing very difficult and costly. For this reason, the CSRC dies have not been exhaustively tested. However, all functionality tested has passed test, thereby indicating a high degree of confidence that the ICs are fully functional.

Although the CSRC die is 610mils x 610 mils, we have tested 66 of the 177 devices received and have seen a 54.55 % yield. Although 111 devices still remain to be tested, it is expected that a similar percentage of "good" dies will result. Once again, this constitutes first pass success on ~9 million transistor, full-custom .35u design. See Figure 3.12.15 for a photograph of the final CSRC device.

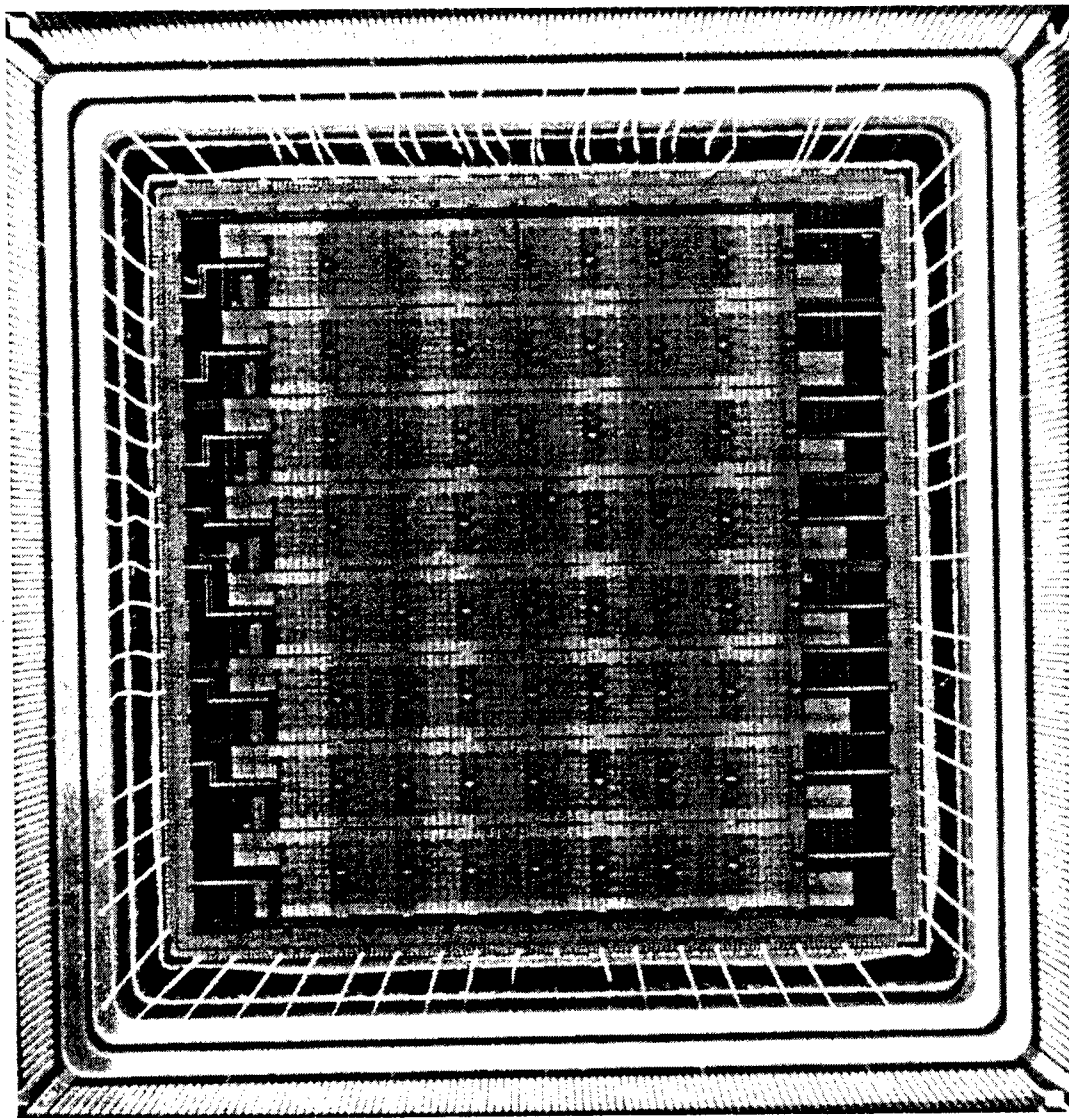


Figure 3.12.15: Die Photograph of Final CSRC IC

3.13 Front-End Design Tools

The overall goal of this research was to achieve an easy to use FPGA development tool which would allow an end user to interactively and expediently implement reconfigurable computing systems for a given hardware architecture. It is believed that this tool should be targeted for the algorithm designer, thereby removing the unnecessary conversion of the algorithm from math to software. It is during this conversion that much of the inherent decomposition potential of an algorithm is lost due to the often-contorting effects induced by attempts to map an algorithm to software for the purpose of simulation. A design tool such as this FCCM mapping software suite has the potential significantly enhance application design productivity.

The algorithm-mapping tool will provide the capability to translate algorithmic elements to synthesizable VHDL or optimized hardware elements which are commonly referred to as macros. The software suite proposed will include the basic tools necessary for the efficient design and implementation of multiple FPGAs by allowing the user to be able to do partitioning across a multi-chip reconfigurable architecture. The proposed FCCM mapping software suite will be comprised of several "plug and play" hardware elements which are optimized for the target reconfigurable devices.

With a system architecture in place, the algorithm designer is free to perform tradeoff studies. The algorithm designer will be able to optimize and adopt an architecture due to the transparent coupling of the hardware to $f(x)$. In essence, an algorithm can be mapped to hardware by a system designer with minimal knowledge of the target hardware, similar to the manner in which a software programmer who uses a high level language can be unaware of how the high level language maps to machine language code.

The approach was to present the algorithm designer with a selection of common, parametrizable building blocks that can be connected in the design space. This research was to leverage the previous Sanders work on FPGA design tools. In particular, the work done at Sanders' with the Ptolemy tool from the University of California at Berkeley will be examined to extract the key algorithmic approaches to pipeline realignment and bitwidth propagation. In addition, Sanders experience with firm macros and the automatic parametrizable generation of the firm macros was to serve as the foundation for the backend tool development effort.

Although this effort received less attention subsequent to Sanders taking on the CSRC IC design and the back-end tool development, two efforts were conducted under this high-level tool effort. The first task performed encompassed the development of a high level tool that is capable of taking a design from a MatLab-like language to silicon. In fact, it was this effort, coupled with Sanders experience that served as the fundamental building blocks for the current DARPA TTO sponsored ACS program (managed by Sanders) entitled Algorithm Analysis and Mapping Additionally. The prototype system developed allows for the real-time instantiation of hardware as is required - similar to the way a personal computer determines whether to execute a particular instruction in microcode or in coprocessor hardware based upon the availability of the function in hardware. The key difference between the developed system and a COTS coprocessor is that the

developed system need not be limited to a finite number of "coprocessor" functions. This is directly applicable to the CSRC technology as the technology being developed will afford clock-cycle reconfiguration of the FPGA's functionality. In other words, "coprocessor" functions can be dynamically swapped in and out of the FPGA on a clock cycle basis thereby enabling the concept of infinite virtual hardware while removing the latency to switch functionality.

The second effort conducted involved the investigation of commercially available design tools and the associated design flows. Five methods for programming reconfigurable computers were investigated: (1) Synopsys VHDL synthesis, (2) Synplicity VHDL synthesis, (3) Schematic entry using the traditional Xilinx tools (M1), (4) Xilinx XDSP tools and (5) Morphologic multi-FPGA design tool. The comparison included VHDL synthesis, designing with vendor blocks, and a multi-FPGA design and partitioning tool. The functions implemented in this test included a simple adder to verify tool flow, a programmable SRAM controller to evaluate the ability to implement control functions and an FIR array filter convolved over an image for clutter suppression to evaluate the ability to implement data path functions.

3.14 Back-End Design Tools

Subsequent to losing Xilinx as a teammate on CSRC, not only did the design of the CSRC IC become Sanders' responsibility, but the burden to develop efficient place and route tools fell upon Sanders as well. Sanders' initial approach was to leverage existing software. The latest version of the University of Toronto's VPR and VPACK software tools were compiled to determine the utility of these tools as it applied to placing and routing the CSRC field-programmable, context-switchable, gate array. VPR is a placement and global/detailed routing tool for array-based FPGAs while VPACK is a technology mapper. The tool accepts a netlist file and an FPGA architecture specification file as its inputs. Figures 3.14.1 and 3.14.2 depict the output of VPR. As can be seen, VPR iteratively routes a given design in an attempt to optimize the device routing. In other words, the tool is an open architecture tool that will place and route a technology-mapped design to a specified architecture. For this reason, it was believed to be a good candidate to be used by CSRC as the place and route tool for the context switchable device being developed. In this manner, the CSRC program would not be burdened by Sanders' inexperience with FPGA toolkit development or the cost of subcontracting. However, the architecture chosen by the CSRC program does not contain segmented routing which precluded the use of the VPR/VPACK software as the place and route toolkit. However, the tool was used to conduct architecture studies and to determine the density of routing resources that should be incorporated in the CSRC FPGA.

Sanders remained in contact with the efforts at the University of Toronto. The possibility of synergistic efforts between the University of Toronto and Sanders were often discussed. Such an effort would have entailed having the University of Toronto add features that both aid the CSRC effort and make sense for general architecture evaluation. Additional tool investigation included the Flow Map tool developed at UCLA, the Emerald tool developed at the University of Washington, and VISSIM developed by Visual Solutions.

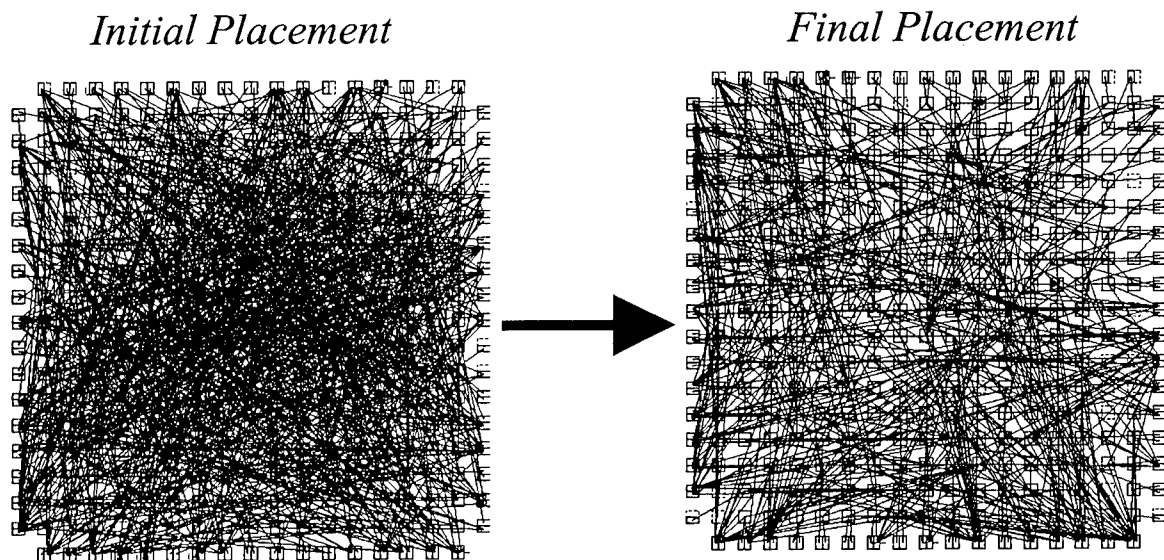


Figure 3.14.1: VPR's Iterative Routing

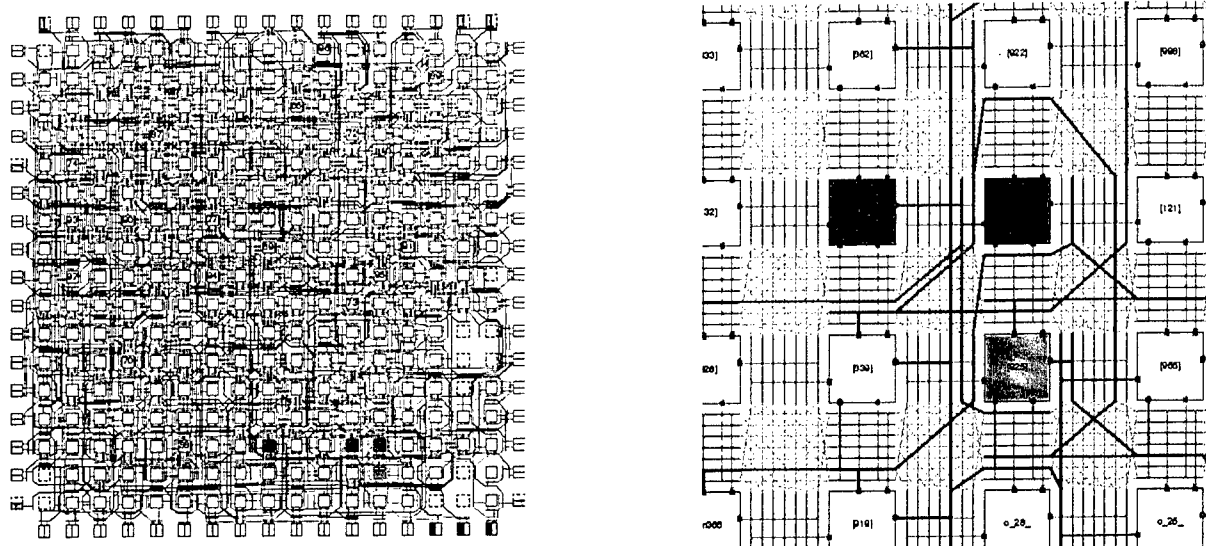


Figure 3.14.2: VPR's Detailed Routing

In light of the fact that the architectural evaluation software search did not lead to a COTS solution to the CSRC toolkit requirement, Sanders outlined a plan for the development of a toolkit that would place, route, and program the device. While attending the kickoff meeting for the *MRC/DSWA Development of a Nonvolatile, Reconfigurable, Radiation Hardened, Field Programmable Gate Array*, Sanders made an important contact with Bill Cox, CEO of FPGA Technologies. (FPGA Technologies is the company that is developing the place, route, and program tools for the Mission Research contract.) Sanders' established a subcontract with FPGA Technologies and worked closely with this company to synergistically and cooperatively develop the toolkit.

As a result of instituting this plan, the back-end software development for the CSRC device has been extremely successful. Almost all of the original goals have been met, and modules have been created according to the original specs. In many cases, additional functionality not originally foreseen has been implemented.

First, a description of completed modules will be discussed. Then, additional modules completed not in the original spec will be described. Finally, the user interface for the tools will be shown, as well as some examples.

Design Database

The first module completed was the design database. The technology mapper, data path generator, placer, router, netlist readers and writers all communicate directly with this database. The database supports timing optimization, incremental compilation, and cross probing within the chip viewer.

The design database module provides an API to the programmer for accessing information about the user's design. All communication with the database will be via

API functions. This provides an abstract layer that hides implementation details of data structures in the database. API functions in the database will be broken into five groups listed in order of importance: netlist functions, library module functions, Boolean equation functions, attribute access functions, and hash table and heap functions. Also incorporated into the database, will be a set of utility functions designed to aid portability between operating systems.

The database had several important effects on the design of the toolkit. It provided a consistent naming convention for representing, accessing, and manipulating data throughout the toolkit. It eliminated the need for intermediate files for tools such as the technology mapper and the placer to communicate with each other. It provides an abstract interface to data that eliminates unimportant data structure implementation details from the user's code. It provides a complete database integrity self check (nlVerifyDesign). It provides a firewall between tools that isolates bugs in one tool from affecting another tool. Perhaps most important, it provides a seamless run-time data structure extension mechanism similar to inheritance. Accessing data in the database through the abstract API is actually faster than accessing data through pointers to C structures.

A case tool called DataDraw was used to draw the schema diagrams for the database and application-specific data extensions. This tool also generated base API functions for accessing and traversing data in the database.

The most important module in the database is the netlist module. This module represents the user's netlist after it has been read into the database, as well as the netlist generated by the technology mapper. The top-level class in the netlist database is the **design** class. It has a list of all netlists in the hierarchy of the design. The **netlist** class represents one VHDL module, or one schematic. Netlists have lists of **instances** and **nets**. Instances connect to nets through **ports**. Equivalence classes (**equiv**) keep track of nets in different netlists performing the same function (with possible inversions). **Trails** between ports describe timing through nets and through instances. **Buses** keep track of groups of nets. Other important classes include the meta-classes (classes describing traits of other classes) **mport** which describes ports on instances, and **mbus** which describes bus-ports on instances.

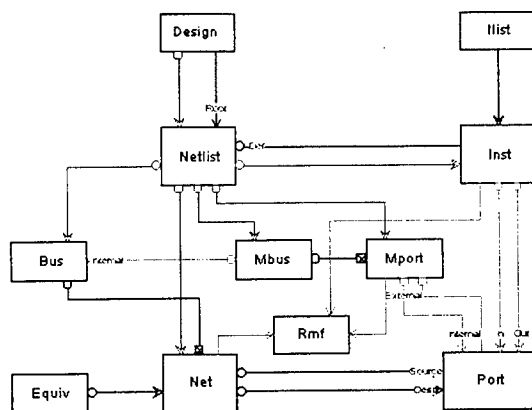


Figure 3.14.3: Database Netlist Schema

Please note that the schemas used in this document conform loosely to UML, which is well documented on the web. Datadraw specific features of the schemas (like the color of relationships which indicate strength) are described in the DataDraw manual on the DataDraw home page.

While the library module originally described in the proposal was implemented, it was eliminated when the netlist database was enhanced to include basic primitive instances. Rather than representing ASIC library cells with the originally described classes such as mod and pin, each ASIC cell is represented as a netlist of primitives. Since the post-synthesis technology mapper is incorporated into toolkit, the ASIC library created consists of only 28 cells. The database, however, is able to support as many library components as desired. The ASIC library is represented with the library design global object (fwLibraryDesign).

The Boolean equation module in the database provides data structures and methods for Boolean equations in two formats: symbolic and Reed-Muller form.

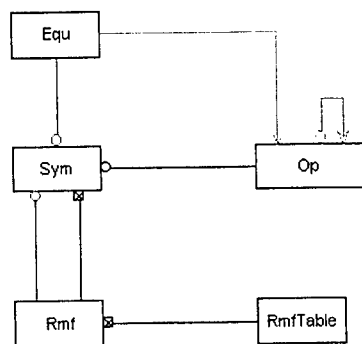


Figure 3.14.4: Logic Equations Schema

The property list section of the database supports generic attributes that can be assigned globally to control tool flow or to specific instances to control mapping or placement. Each class that has a property list is shown in the following schema.

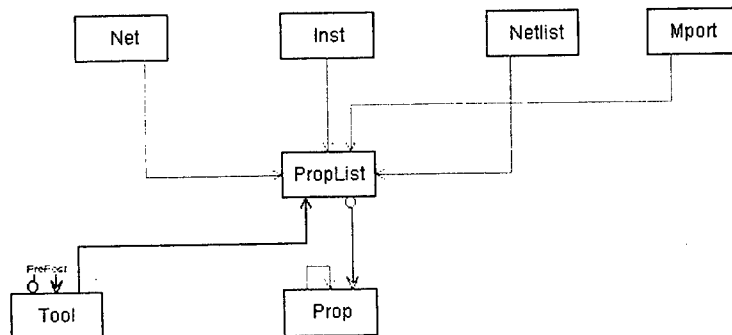


Figure 3.14.5: Global Property Assignment

The heap and hash table module will provide generic hash tables and binary heaps.

The timing schema shows the relationships between ports and trails. The **domain** class describes the timing of clock domains, and has a list of nets assigned to the domain. Table, row, and delay classes hold data from the csrc.tim file, which completely specifies the timing of CSRC devices.

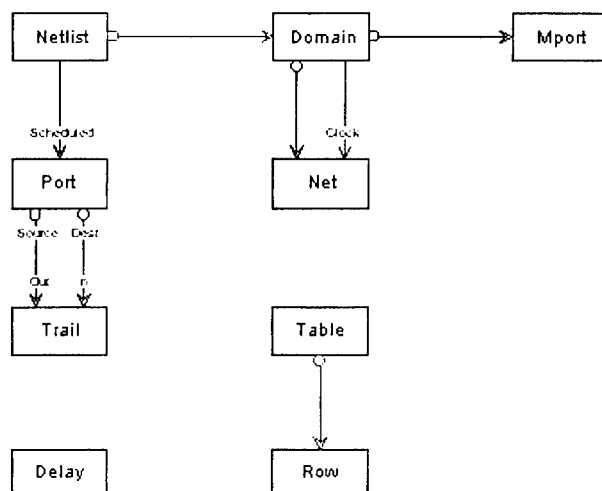


Figure 3.14.6: Timing Schema

Structural Netlist Readers and Writers

Structural Verilog and VHDL netlist readers were implemented to read technology mapped designs into the database. Structural Verilog and VHDL writers were also written to verify the netlist reader and provide back annotation. SDF timing files as well as VITAL compliant writers have been written. VITAL compliance was added as a reasonable extension to the original proposal.

This module reads a subset of Verilog (VHDL) used by Synopsys and Synplicity tools when writing structural netlists. Netlists may be hierarchical. A netlist flattener was written to create a flat netlist in memory before running the technology mapper.

Physical Database

To complete the information in the database needed by the placer, router, and bit-stream generator, the FPGA physical description was created to describe the logic cell and routing structures in all contexts. Rather than write C-code generators for the database as originally proposed, an ASCII technology file configures all aspects of the physical database, including the routing graph. The technology file is created from the Architecture Editor, described later.

The physical database represents the architecture of the chip. It was implemented before the placer or router, and both use it. The top will class is the **die** class. The **package** class describes bonding of pads and pins. The **carry** class describes cell arrays. The **row**

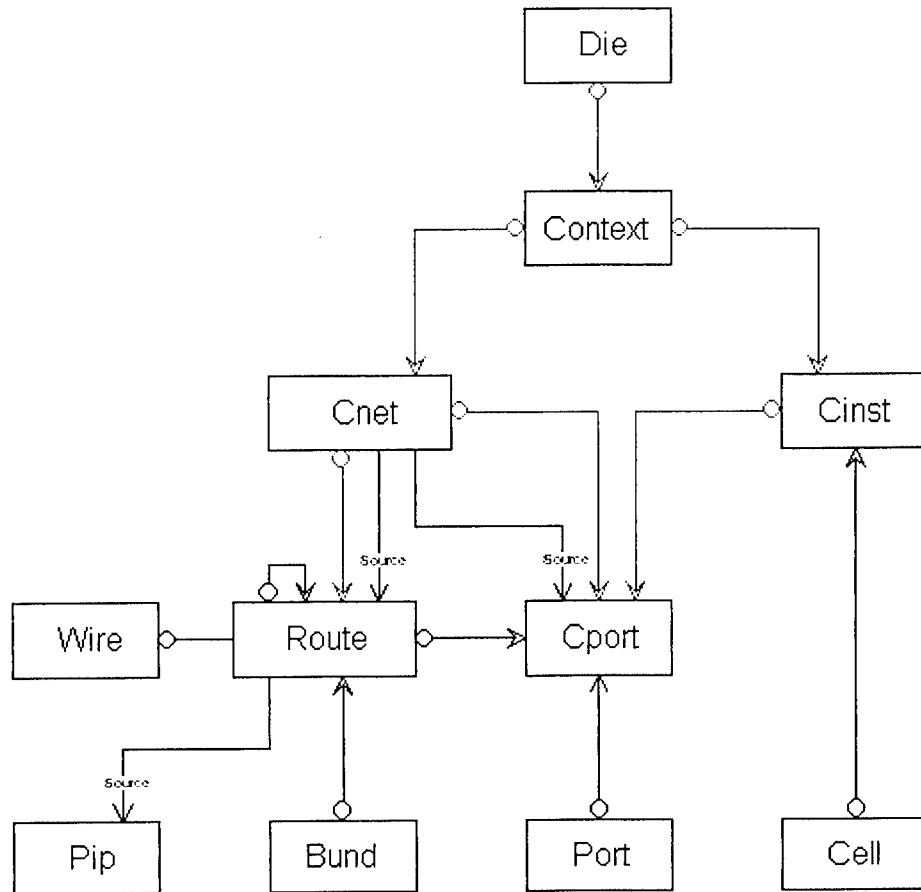


Figure 3.14.8: Context Class

FPGA Viewer

A viewer was developed to show users the FPGA routing resources and placed and routed designs. The first use of this module was to verify the output of the Architecture Editor (technology file). Later, it was used to verify the results place and route. This interface is available on the PC platform, but is easily ported to Unix using the WinMain porting toolkit. Porting of this interface is not included, due to the cost of the WinMain tools. However, a TCL command line interface is available on both PC and Unix platforms.

The FPGA viewer has several purposes. It helps to verify results of place and route. The end-user is able to use this tool in order to gain confidence in the automatic tools, since they can easily examine the tools' results. It also helps early users to discover bugs in the placer or router that may exist. If a user's design does not meet his required timing, the viewer is handy in determining how the design could possibly be made faster. Critical instances are highlighted in blue in the view.

Views of the user interface are shown later.

Automatic Placer

A placer was written next. Users are expected to assign hierarchical blocks of logic to contexts manually. There is a mechanism for users to manually place logic to specific locations within a context using attributes on instances. Simulated annealing is used to simultaneously place multiple contexts, taking into account the global routing structure. The significance of the placing all contexts that will be sharing data simultaneously is tremendous due to the co-location requirement imposed on CSFFs that wish to share data. Without this capability, each additional context would impose additive placement constraints on the next context. The additive nature of these constraints would make placement of subsequent contexts increasingly more challenging and quickly impossible. Because the routing is hierarchical in nature, the placer is able to determine the global routing. This leads to excellent timing estimates during placement, as well as good routing completion rates. The first version of the placer was not timing driven, but was enhanced later. Incremental capabilities were also added later.

While algorithms used in the placer are fairly simple, performance of the placer is central to the quality of the toolkit. A significant effort was made to insure run times are acceptable.

The placer was made timing driven by incrementally updating the timing graph after every move. The incremental capability simply allows the placement of all flip-flop and pad instances whose functionality has not changed since the last compilation to remain in the same place as before. This greatly speeds the run-time of the placer, although lower quality of results may occur if many incremental changes are made.

Router

Because the placer is sometimes unable to resolve complicated routing using its simple routing heuristic's, a simultaneous global and detail router was written to complete the job. This router understands details of the routing architecture, and is capable of making de-tours to reach completion. Again, because of the regular routing structure, routing is accurate and fast. The router is based on traditional maze algorithms, expanding from drivers to destinations. Rip up and re-route algorithms are used to complete routing in highly congested areas. As a last gasp, buffers are inserted to help routing.

First, the router analyzes routing resources in the physical database to determine commonality between wires. Wires with identical destinations are grouped into bundles. Adjacencies between bundles are then built. This provides the maze on which maze expansion is performed. Since the bundles may not reach all the wires in adjacent bundle, the adjacencies contain bit masks showing which wires are reachable. During maze expansion, each bundle keeps a mask of wires currently reached. When an expansion wave reaches a bundle, it first checks to see if it reaches any un-reached wires. Expansion continues only while new wires are reached.

Users are able to see with the chip viewer that the router has done a proper job of completing routing and minimizing delays. In cases where the router is unable to complete, problem areas are highlighted. It is expected that users will not feel any need

for a manual routing editor. The run-time of the router is substantially less than that of the placer.

Automated Testing

To verify the correctness of the place and route toolkit, a simple functional simulator automatically verifies that each placed and routed design performs the same function as the original physical netlist when simulated against several random vectors. This approach catches well over 90 percent of all functional errors introduced by bugs the toolkit. The value of this is hard to over-emphasize. This toolkit is very complex, with over 150,000 thousands of lines of code. Combined with the regression test suite of designs which are automatically run with the batch TCL interface, high confidence that logical errors are not introduced by the tools is achieved.

This feature is left on all the time. Each design the user creates must pass this self check. The users original netlist remains in memory once read in. A new netlist is created from the routed design in the physical database. These two netlists will be compared by loading random values into input pads and flip-flops, and then simulating the netlists. Values on equivalent nets in the two netlists must be the same to pass.

Bitstream Generator

A bitstream generator was written to generate programming data for the FPGA. Sanders developed Perl scripts that convert an ASCII script into a binary input stream to configure the contexts. This format provides a human readable file that can be examined for errors manually. The perl scripts are run automatically from the user interface after routing completes

Timing Analyzer

The final tool to be developed was a timing analyzer that estimates routing delays. SDF (Standard Delay Format) is written out so that Verilog and VHDL simulators can perform timing simulations of routed designs. A crucial part of timing analysis is good timing estimates of routing delays. A timing file, `csrc.tim`, provides Sanders with the ability to accurately describe all of the delays in the device.

The SDF file describes rising and falling delays from each input of each instance to each output. Routing delays are pushed onto inputs of the instances driven by nets. Setup and hold delays are also described.

Once the timing analyzer was running, timing simulations were manually checked for correctness.

Timing Driven Enhancements

Users will create TCL scripts describing their timing requirements. In particular, there are the following commands:

```
clock_period mportName period
main_clock_period period
pad_arrival mportName arrival
pad_departure mportName departure
pad_setup mportName setup
clock_to_out mportName delay
```

During placement, all defined clock domains are incrementally checked for critical paths, and nets not meeting timing add a penalty to the placement cost. All paths are thus simultaneously optimized during placement.

Incremental Capabilities

A highly desirable feature is the ability to make a small change to a design without having to place and route from scratch. After timing driven enhancements were made, the toolkit was enhanced to recognize differences between successive netlists read in the toolkit. When they are similar enough, the user is allowed to specify an incremental place and route in the options dialog box. In this mode, all of the placement information of pads and flip-flops, which have not changed, are used. New or different instances are placed in locations that were previously unused.

Architecture Editor

The most significant additional module developed, but not originally specified, was the Architecture Editor. This tool provides a Windows user interface for drawing all of the FPGAs logic cells and routing resources. Once the device has been drawn, a technology file can be exported which is used to initialize the physical database.

The benefit of the Architecture Editor is flexibility. Many changes were made to the architecture during development, and the Editor allowed quick and easy adaptation to these changes. Even the task of drawing the device in the viewer would have been difficult without the Editor.

The main window looks like:

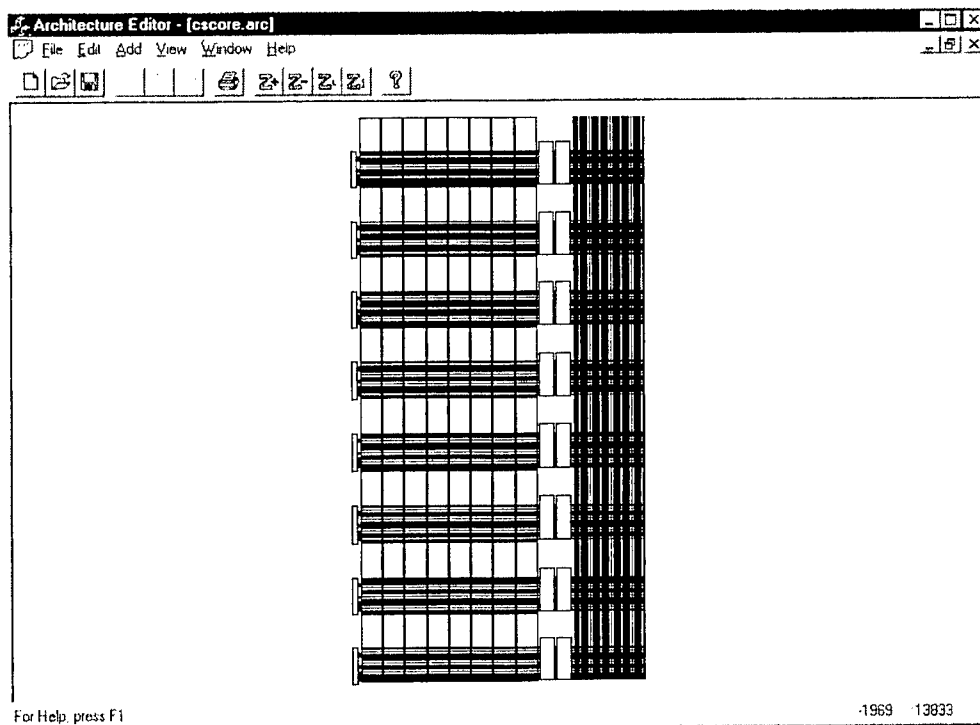


Figure 3.14.9: Architecture Editor Main Window

This shows the top-level view of the CSRC device. As in the Chip Viewer, the user can zoom in and out, but in addition, he can select sub-blocks of the device and edit them. With a couple of mouse clicks, the above device was modified to look like:

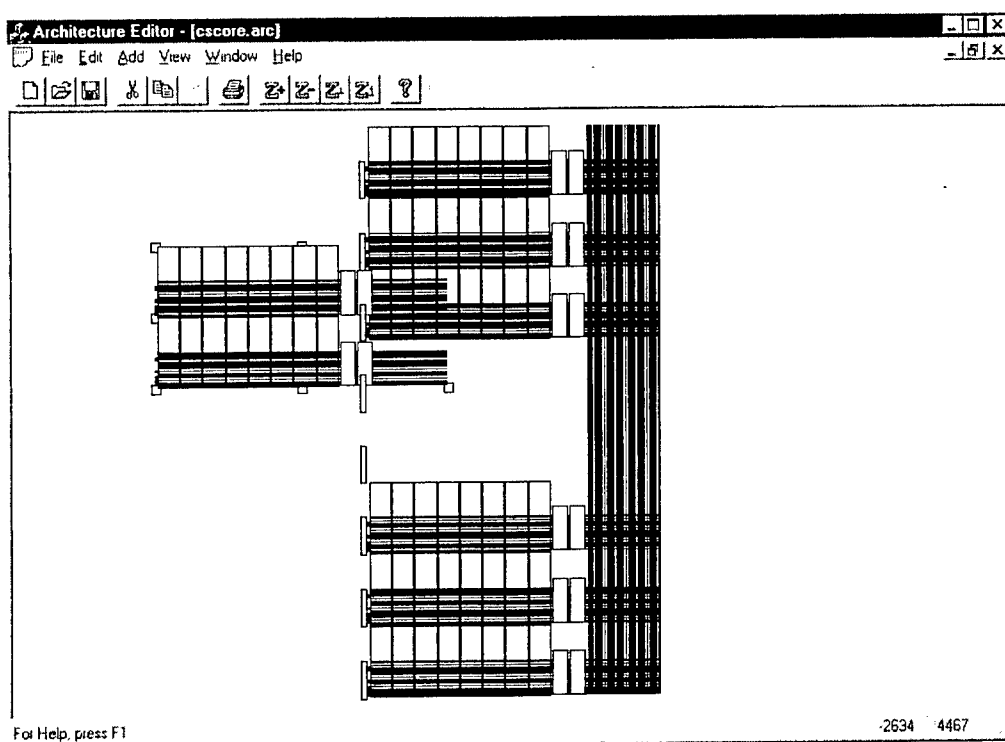


Figure 3.14.10: Architecture Editor Edits

Infinite undo/redo is supported in the Editor. Operations are optimized for automating the most common tasks. For example, repeated structures can be placed in regular rows or columns with the duplicate command. Wires can be resized in groups. Names can be applied to objects with a single mouse click.

The Architecture Editor makes FPGA Technologies' tools the most flexible FPGA place and route system ever designed. Support for new architectures is far less difficult than in any other system.

User Interface

The main window at startup consists of the project dialog box and the main menu.

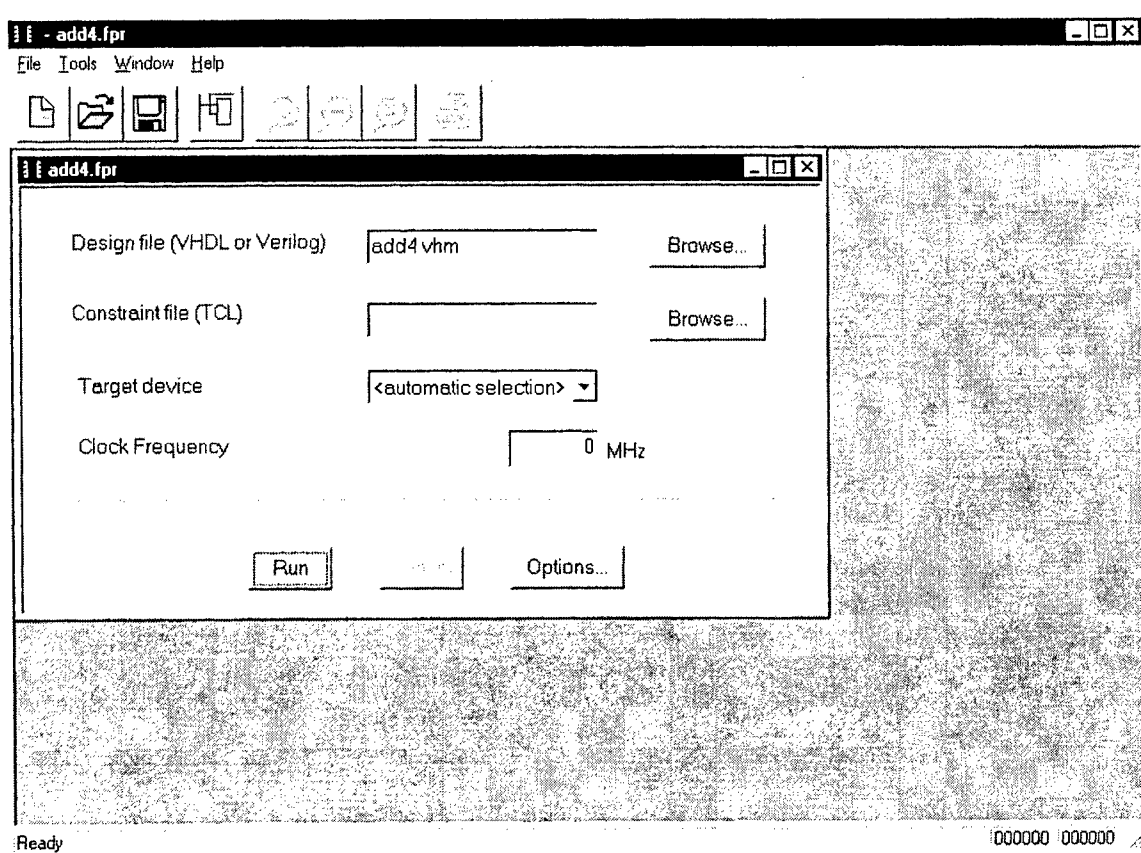


Figure 3.14.11. User Interface Main Menu

The user can type the name of his structural netlist, or click browse to have a file dialog box appear. The constraint file is a user specified TCL script that specifies multiple designs for multiple contexts, pad placements, and timing constraints. The clock frequency can be entered here rather than in the TCL file. While there is currently only one CSRC device available, support has been added for automatically selecting the best device for when multiple devices exist. The Options dialog box is reached by clicking the Options button in the project window.

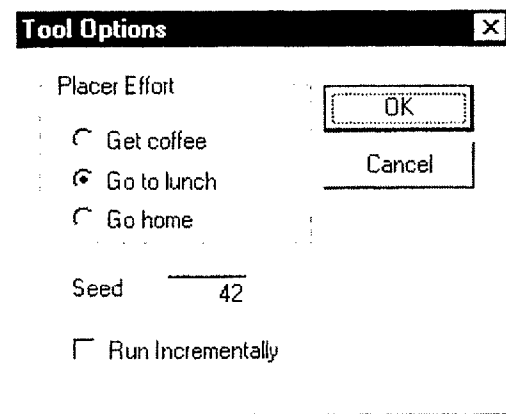


Figure 3.14.12: Tool Options

Options currently supported are placement effort and random seed, and whether or not to use previous placement information for pads and flip-flops.

The toolbar at the top allows quick access to zooming functions, project load and save, and loading the previous place and route results. From the Tools menu, two functions are available:

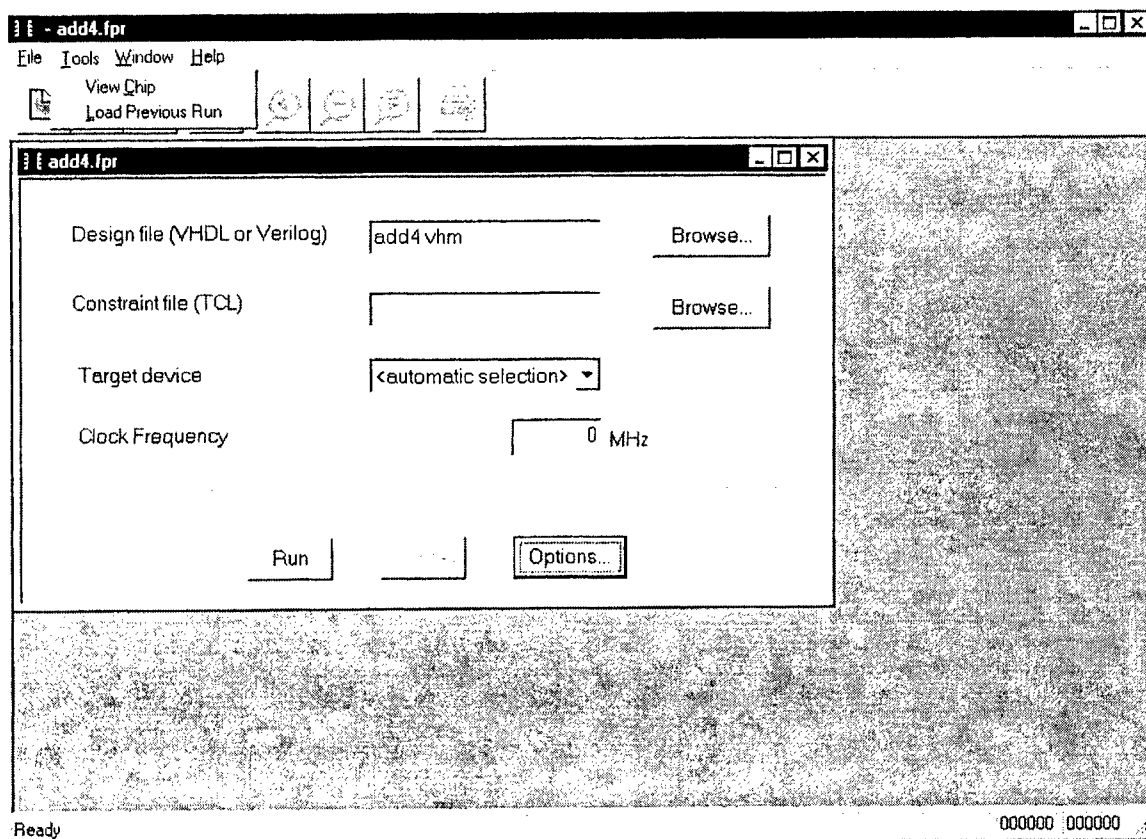


Figure 3.14.13: Pull-Down Menus

View chip allows all of the devices wires and logic elements to be examined without having to place and route any design:

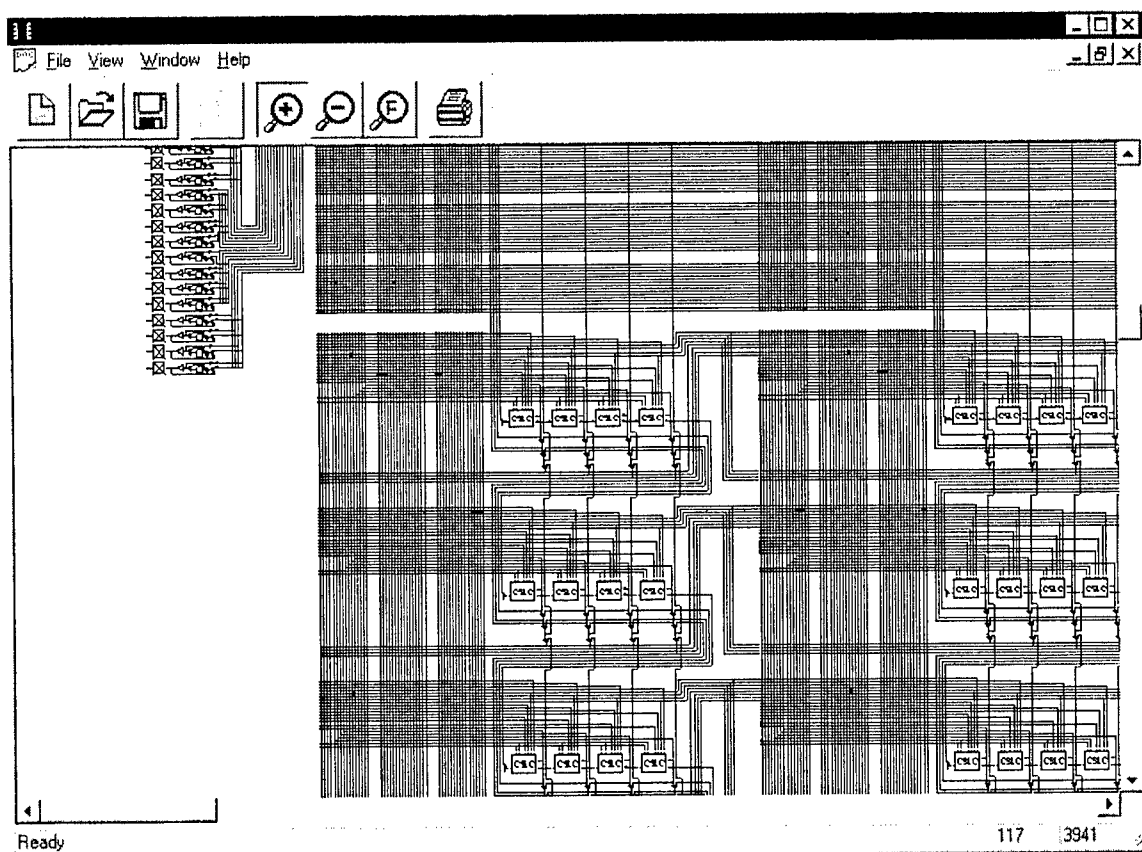


Figure 3.14.14: FPGA Resource View

The 'Load Previous Run' menu item functions the same as the toolbar short-cut in that it loads the results of the previous place and route.

By pressing the 'Run' button, all of the automatic tools are run on the users design. Users do not need to be aware of all that is going on when this is done, but steps that are included are:

- Reading the user's VHDL netlists
- Running any TCL constraint file specified
- Technology mapping the design
- Creating schematics for post-tech-mapped netlists for Synplify
- Selecting the smallest device that will hold the mapped design
- Initializing the physical database from the technology file
- Loading package information
- Loading timing information
- Loading the previous placement information if incremental is selected
- Placing the contexts
- Routing the contexts

- Calculating post-routing delays
- Determine critical paths
- Write VITAL compliant VHDL and SDF
- Simulate design on random vectors to determine logical correctness
- Generate bitstream file, and launch Perl scripts
- Save the new placed and routed design to a .par file
- Generate the view of the placed and routed design

After routing completes, the user can zoom in on his design to examine results:

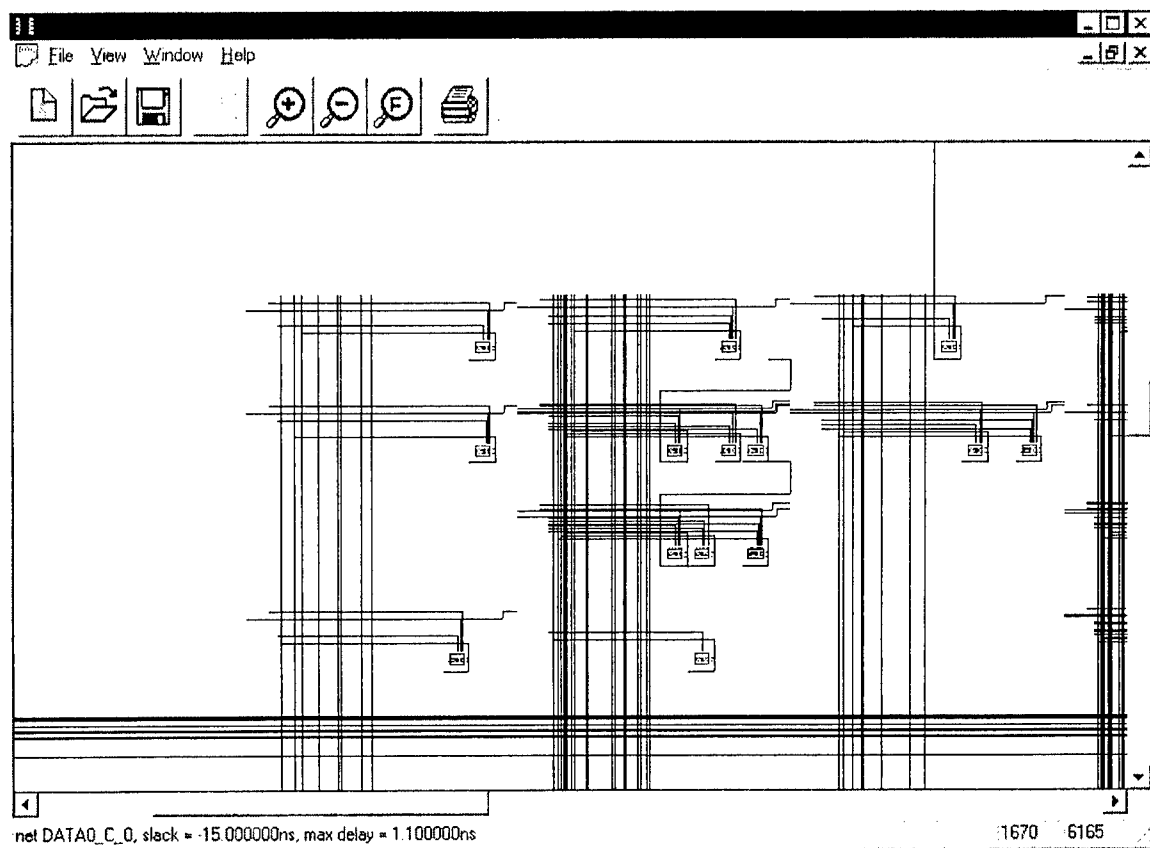


Figure 3.14.15: FPGA Post Place and Rout Viewer

Clicking on them can highlight nets. Clicking on instances highlights the instance in each context where it exists.

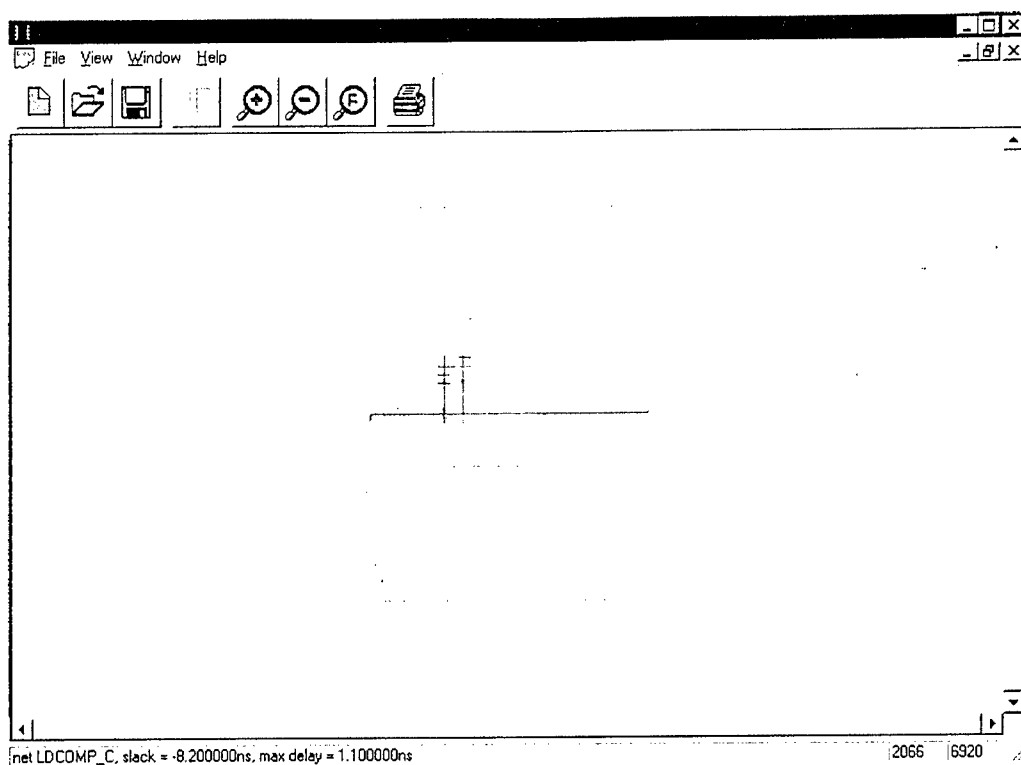


Figure 3.14.16: Net Highlighting

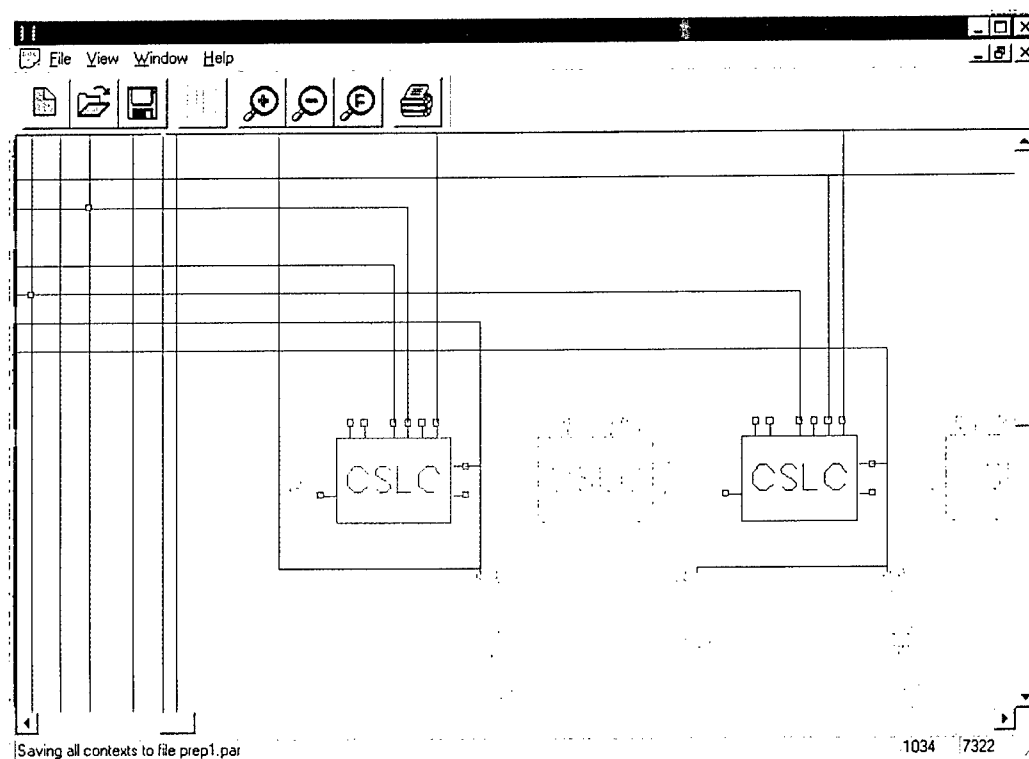


Figure 3.14.17: Critical Nets are Highlighted in Blue

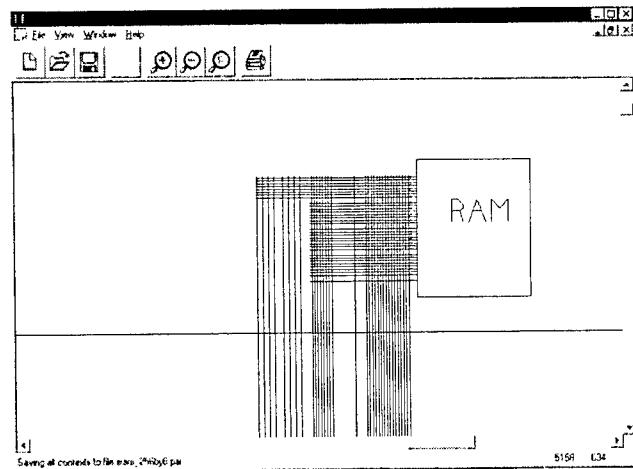


Figure 3.14.18: Block RAMs are Fully Supported

There are also behind-the-scene features. The batch version is now available and was used to run regression tests before each new tool release. There are new configuration files: `csrc.tim` accurately all timing estimates for the device. `csrc.pkg` specifies package to pad bonding. Pad pre-placement is supported, and takes pin names rather than internal cell names. VITAL compliant SDF and VHDL files are created post-routing to allow accurate timing simulation. Data sharing between public registers A and B is automatically inferred when common net names are used in multiple contexts. The router has been enhanced to insert buffers if needed to complete routing. Synplicity schematic files are now created after tech-mapping so that the user can see how the design was mapped within the Synplify Analyst. Load-times for the technology file have been cut in half with a faster reader.

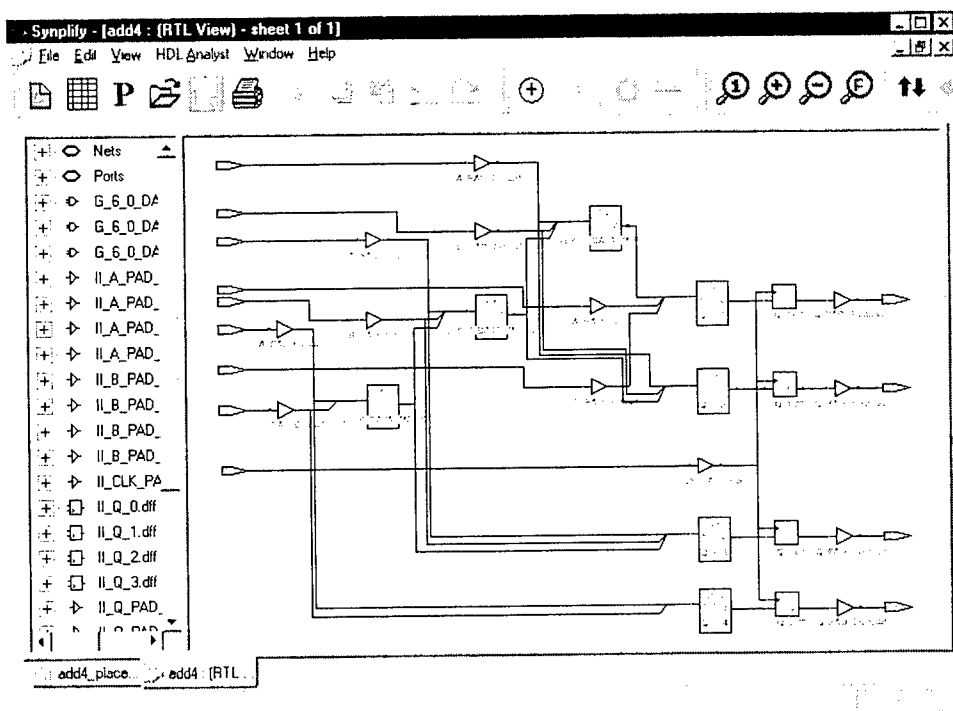


Figure 3.14.19: Integration with Synplicity Analyst

TCL Commands

The TCL interpreter has been integrated to provide a batch mode interface. A complete list of commands is summarized below:

```
compare_designs
    [ returns int ]
    Used to verify that the tools have implemented the logic of the original design correctly.

read_verilog fileName
    [ returns int ]
    Reads a structural Verilog netlist to be implemented in a context.

write_verilog fileName
    [ returns int ]
    Writes a structural Verilog netlist.

read_vhdl fileName
    [ returns int ]
    Reads a structural VHDL netlist to be implemented in a context.

write_vhdl fileName
    [ returns int ]
    Writes a structural VHDL netlist.

write_srs fileName
    [ returns int ]
    Writes a Synplify compatible schematic file of the design.

techmap
    [ returns int ]
    Technology maps the design.

load_techfile
    [ returns int ]
    Reads the csrc.tec technology file.

place
    [ returns int ]
    Places all contexts.

route
    [ returns int ]
    Routes all contexts.

location netName cellName
    [ returns void ]
    Preplacement command for nets or top level ports.

inst_loc instName cellName
    [ returns void ]
    Preplacement command for instances.

net_loc netName cellName
    [ returns void ]
    Preplacement command for nets.
```

```

port_loc portName cellName
    [ returns void ]
    Preplacement command for top level ports.

generate_bitstream fileName
    [ returns int ]
    Generates the .csrc ASCII bitstream and calls perl scripts to create the .bin binary file.

load_package
    [ returns int ]
    Loads package info for the device from csrc.pkg.

load_timing
    [ returns int ]
    Loads timing info for the device from csrc.tim.

save fileName
    [ returns int ]
    Saves the place and route result to a .par file.

reset
    [ returns void ]
    Resets the toolkit so that another design can be run.

set_seed seed
    [ returns void ]
    Sets the placer random seed.

set_effort effort
    [ returns void ]
    Sets placement effort. 1 is low, 2 med, and 3 is high.

calculate_net_delays
    [ returns void ]
    Runs the net delay extractor.

analyze_timing
    [ returns double ]
    Calculates critical path delays.

write_sdf fileName
    [ returns int ]
    Writes an SDF delay file for use with a VITAL compliant VHDL simulation.

write_vital fileName
    [ returns int ]
    Writes a structural VHDL (VITAL compliant) netlist for use in post-routing timing simulation.

select_default_part
    [ returns void ]
    Determines which part best fits the design, and selects it as a target for place and route.

clock_period mportName period
    [ returns void ]
    For use in multi-clock environments. Sets the period of a specific clock.

```

`main_clock_period period`
[returns void]

For use in single clock environments. Sets the global clock period.

`pad_arrival mportName arrival`
[returns void]

Sets expected arrival time of a signal at a pad before the next clock edge.

`pad_departure mportName departure`
[returns void]

Sets the expected delay after the previous clock edge by for when a signal must be valid.

`pad_setup mportName setup`
[returns void]

Sets the expected time that a signal is valid at an input pad before the next clock edge.

`clock_to_out mportName delay`
[returns void]

Sets the required clock-to-out timing at a pad.

`load_guide_file fileName`
[returns int]

Loads the previous place and route results so that placement information of pads and flip-flops can be reused.

3.15 Reconfigurable Computing Module (RCM)

The Reconfigurable Computing Module's (RCM) primary objectives were to provide hardware to demonstrate the operation of the Context Switching Reconfigurable Computing (CSRC) device and to be a commercially viable processor with on board reconfigurable and context switching logic. The architecture of the RCM provides good general use and extensive flexibility in the configurations. See Figure 3.15.1.

The Reconfigurable Computing Module form factor was required to be on a commercial standard that would allow connection into a commercially available host computer system. The PCI long card form factor was selected because of its high performance, capabilities for full concurrency with processor/memories subsystems, ease of use and support of multiple families of processors as well as future generations of processors (by bridges or by direct integration).

Requirements on the RCM called for a node processor to be modern, main line with floating-point capability and extensive software development tool support. The processor selected was the MPC750 RISC Microprocessor. The MPC750 is targeted for low-cost and low-power systems and consists of a processor core and an internal L2 Tag combined with a dedicated L2 cache interface and a 60x bus. The PowerPC 750 microprocessors are super-scalar, capable of issuing three instructions per clock cycle into six independent execution units: two integer units, floating point unit, branch processing unit, load/store unit and system register unit. The ability to execute multiple instructions in parallel, to pipeline instructions, and the use of simple instructions with rapid execution times yields maximum efficiency and throughput for PowerPC 750 systems.

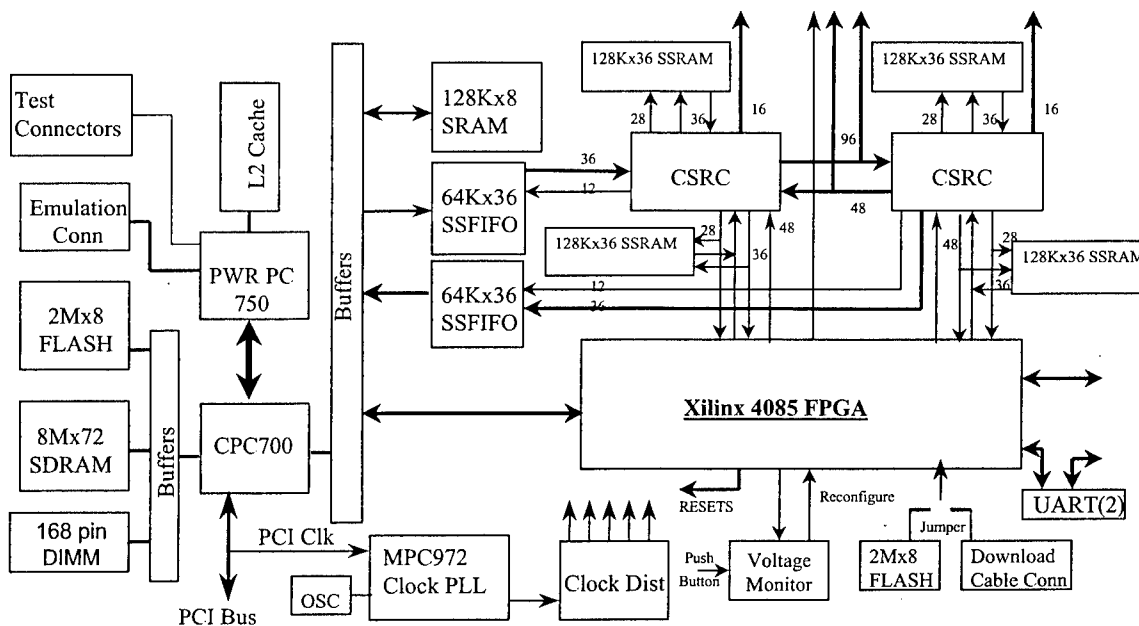


Figure 3.15.1: RCM Block Diagram

The processor is rated to operate at a core clock rate of 300 MHz and a bus clock ranging from 25 to 83.3 MHz. The MPC750 processor has complete support for a private L2 cache. A pair of 128K x 36 Synchronous SRAMS is connected to provide 1 Mbyte of cache with byte parity error detection capability. The clock speed of the L2 cache interface is controlled by the processor, it is synchronous with the CPU core and can be set to core L2 ratios of: 1, 1.5, 2, 2.5 or 3.1.

A CPC700 device provides the connection between the PCI bus and the processor. The CPC700 provides a PowerPC common hardware reference platform (CHRP) compliant bridge between the PowerPC microprocessor and the PCI bus. The CPC700 integrates secondary cache control and a high performance memory controller. The CPC700 provides an integrated high-bandwidth, high-performance, TTL-compatible interface between a 60x processor, a secondary (L2) cache or additional 60x processors, the PCI bus and main memory. The initiator and target PCI interface is 32 bit wide and PCI 2.1 compliant.

The RCM's main memory is composed of SDRAM's arranged in up to 4 banks. Bank 0 consists of a 64Mbyte arranged as 8M x 64. There is one 168-pin DIMM connector to accept commercially available SDRAM modules. The main memory is controlled by the CPC700. The MPC106 provides byte parity. When less than the full 64 bit data bus is written with parity operating, the 106 registers the write data, reads the addressed 64 bit wide location. It replaces the read data with the appropriate write data and recalculates the error control bits on the full 64 bits before it writes to RAM.

The processor to CSRC communications is primarily through two sets of FIFOs that are 36 bits wide. The depth is dependent on the version of the RCM, the FIFO devices are pin compatible to support anywhere from 8K deep to 64K deep. The FIFOs have configurable almost full and almost empty flags. These flags are inputs to the support FPGA, which make the flag values available to the processor or CSRC devices. The FPGA could be configured to generate interrupts on flag conditions or just to provide flag status.

The RCM supports two CSRC devices in 560 BGA packages. Each device is connected to private SSRAM (128k x 36). There is another 128K x 8 SRAM that is shared with the support FPGA. The 64 signal lines connected to the SRAM and FPGA are general purpose and may be used for shared control of the SRAM or for communications between the CSRC and the FPGA, or a mix. The two CSRC devices are directly connected with 144 signal lines and there are 48 signal lines from each CSRC to the support FPGA. CSRC1 is connected to a 36 bit wide FIFO to the processor bus. The assumed data flow is thus from the processor through the CSRC1 to CSRC2 and back to the processor. If both devices need to send data to the node processor, the CSRC1 sends its data to CSRC2. CSRC1 is in control of this FIFO interface and when there is insufficient data, it will let the pipeline flush and suspend operations until new data is available. If there is a need for "direct access" to the main memory from the CSRCs, the more than one hundred connections between each CSRC and the FPGA can be used with an appropriate FPGA design to allow the CSRC to gain access to the processor bus.

Either the FPGA provides clock domain translation or the CSRC operates synchronously with the processor section. The CSRC devices are configured via the support FPGA. The data may come from the host processor via the PCI bus, the node processor or from the configuration FLASH attached to the FPGA.

The Xilinx FPGA is intended to provide a variety of support functions. It is packaged in a 560 BGA. The FPGA contains as a minimum the ability to receive interrupt requests from the host processor, manage FIFO control flags, program the CSRC devices, and serve as a DMA controller to move data to and from the CSRC devices.

In Summary, the key identifiable features of the RCM board are as follows.

- 64 Mbytes SDRAM
- Up to 128Mbytes DIMM
- 2 Independent memory banks per CSRC
- Input & Output FIFOs (16k x 36)
- 300MHz PPC750
- Xilinx 4085
- 66MHz Local Bus
- 32k bytes each (instruction and data) L1 cache
- 1 Mbyte L2 cache
- 2 CSRC ICs
- Windows NT limitation - Only 128MB accessible from PC - Not true with Linux
- PPC has access to all memory space
- Xilinx, FIFOs, DIMM, SDRAM, SRAM memory mapped from PPC750

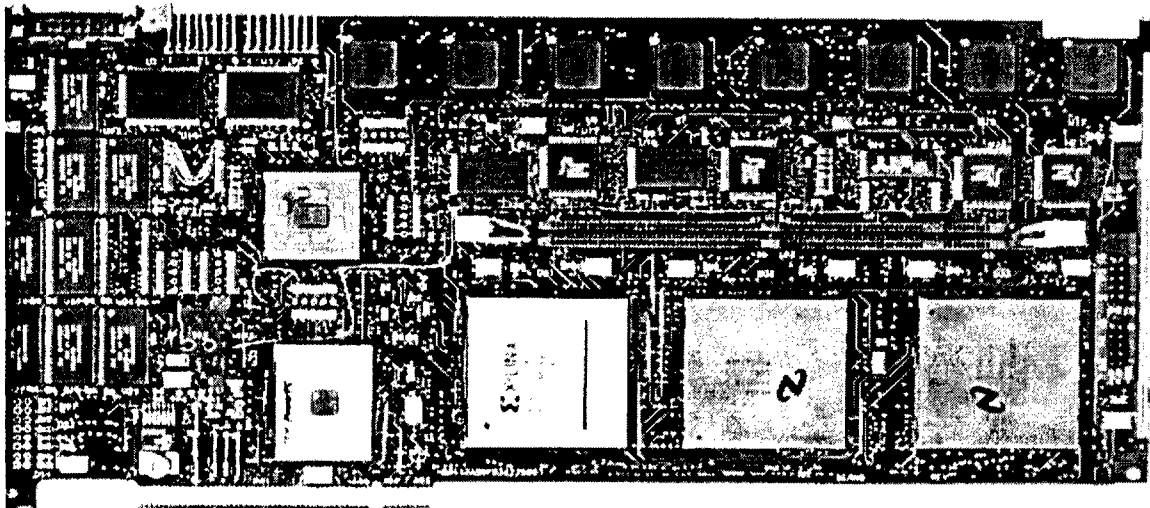


Figure 3.15.2: RCM Circuit Card

RCM Status

- The board including the boot code is operational. The RCM board is recognized by the host PC and data can flow from the host to the RCM board (and visa versa). In addition, the boot code has been augmented to afford the developer MetroTRK

debugger support. All of the resources on the board are memory mapped via the CPC700 (8Mx72 SDRAM, 168 pin DIMM, Xilinx, FIFOs, and 128kx8 SRAM). In fact, the host PC can directly write to any of the memory mapped elements. A photograph of the RCM board can be seen in Figure 3.15.2. The boot code for the RCM:

- programs all control registers in the PowerPC
- programs registers in CPC700 to configure memory spaces unavailable upon power up
- programs CPC700 registers defining PCI interface
- programs CPC700 registers configuring internal peripherals i.e. UARTs, I2C controllers, etc.
- and eventually passes the control to the main() function

PPC750 Development Software

The software to develop the PPC code required an embedded software compiler. An evaluation of PowerPC development tools was completed and a decision to use the CodeWarrior cross-development tools for the PowerPC 750 processor was made for the following reasons:

- UNIX based GNU tools don't seem to support PPC750; trying to compile to a different version of the processor might have worked but there is certain level of uncertainty with this – and, we wouldn't be able take advantage of specific features of PPC750
- Solid and well supported product
- Integrated Development Environment to ease learning process and allow seamless cooperation between different parts of the system – project gets compiled, assembled and linked by clicking on just one button
- Standard features like inline and stand-alone assembly etc.
- Provides upgrade path for PC only development system
- Relatively inexpensive
- Provides users debugging capability without necessity of buying debugging hardware (the latter is much better solution but costly) – so called Target Resident Kernel is included with the package; this is software debug monitor
- Generates object files in ELF/DWARF format – this format is supported by most debuggers available on the market; in particular we can and will use SDS debugger along with HP logic analyzer/emulation probe to initially debug the hardware and start embedded software development to deliver the system to Georgia Tech for further development

3.16 RCM Board Support Software

The WindowsNT based board support software was designed such that an SRECORD is used to download the PPC SW from the PC and a starting address is given to the PPC to start its execution from. This methodology facilitates putting any operating system or code on the PPC. The data transfer routines between each platform have been created. In fact, it is possible to play data files through the RCM board from PC (file)→RCM memory→PPC→Xilinx→CSRC A→CSRC B→Xilinx→PPC→RCM memory→host PC (file). The user can configure both the Xilinx and the CSRC devices, as well as reset the CSRC devices. The board support package maintains the filenames of download contexts to CSRCs and can manage switching of contexts. Essentially, this board support package was designed to enable a new RCM user to get up and running fast while affording the ability to “touch” all components on the RCM from a simple GUI. The GUI for the board support package is shown in Figure 3.16.1.

The screenshot displays the 'GUI for Board Support Package' with several sections:

- Configuration/Context Switching:**
 - Checkboxes for configuring Xilinx, CSRC A, and CSRC B with specific files. Each has an 'Open' button.
 - Buttons for 'View Docs' and 'GO'.
 - Section for 'Configure CSRC A' and 'Configure CSRC B' with context lists (0-3) and 'Not programmed' status.
 - Checkboxes for 'Switch CSRC A to' and 'Switch CSRC B to' with context lists.
 - 'Current Context' indicators for CSRC A (0) and CSRC B (1).
- Load/Execute PowerPC code:**
 - 'S-record file:' with an 'Open' button.
 - 'Execute @ start address (hex):' with a 'GO' button.
- PowerPC configuration code:**
 - 'S-record file:' with an 'Open' button.
- Read Data File:**
 - 'Data File:' with an 'Open' button.
 - 'Read!' and 'Process!' buttons.
- Board Status:**
 - 'Reset CSRC A' and 'Reset CSRC B' buttons.
 - 'Exit' button.
 - Status text: 'Data is finished processing.... Ready.'

Figure 3.16.1: GUI for Board Support Package

3.17 RCM Demonstrations

A nation critical Ultra-Wide Band (UWB) Synthetic Aperture Radar (SAR) algorithm was selected for demonstration on our CSRC Reconfigurable Module (RCM). Foliage Penetrating (FOPEN) and Ground Penetration (GPEN) applications require low artifact image formation. In Unmanned Aerial Vehicle (UAV) applications the amount of available signal processing is limited by the size, weight and power constraints of the platform. One of the challenges to ARL is in implementing algorithms efficiently on the desired surveillance or weapons platform. For example, a tactical UAV may have a few hundred watts of power, a cubic foot of volume and a 50-75 pound payload restriction. Figure 3.17.1 illustrates several technology transition opportunities that would be afforded as a result of accelerating ultra wideband SAR image formation. In fact, this work that the CSRC program conducted in cooperation with ARL was submitted by ARL as an ARL "Top Deliverable". Therefore, one of the goals for this work is to explore implementations that combine algorithms and architectures to address these stringent constraints. Scale Recursive Image Formation promises to save computation cycles by using previously computed results to generate the next level of resolution and being able to stop the process when the image is "good" enough and thereby reduce the signal processing burden. Adaptive Computing promises to provide improvements over general purpose digital signal processors by a factor of 10 or more, but the improvement factor depends on the algorithm and data flow to be implemented.

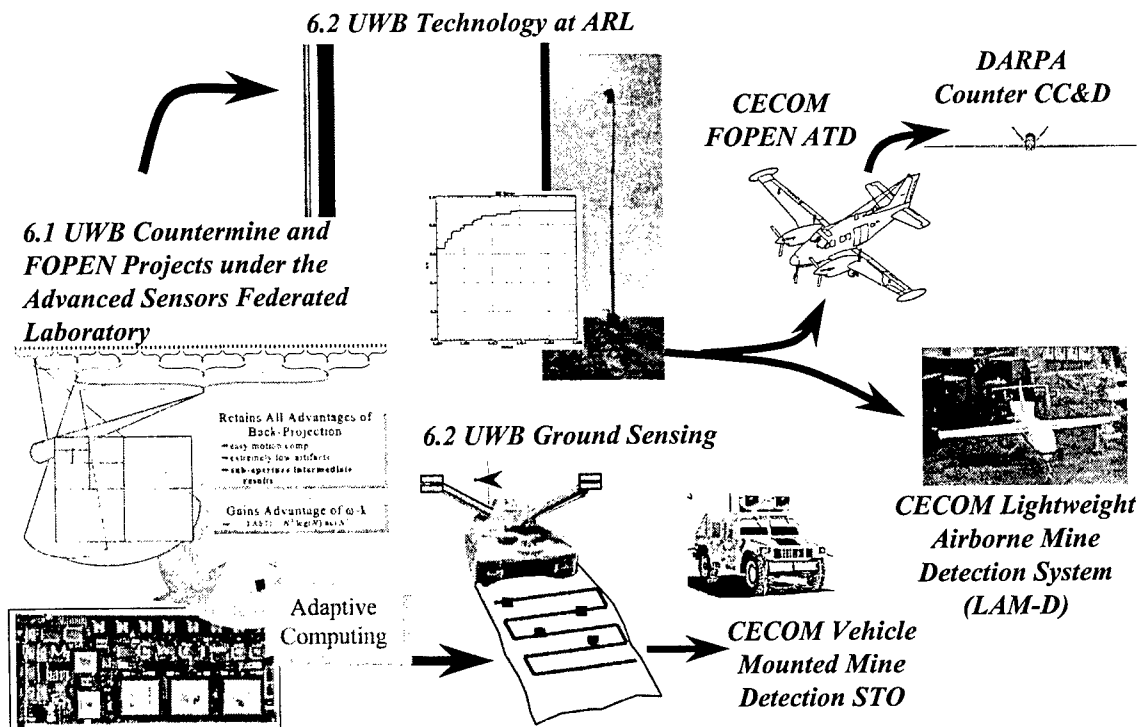


Figure 3.17.1: Wideband SAR Technology Transitions

Specifically, we plan to demonstrate the Quadtree algorithm as depicted in Figure 3.17.2.

This algorithm has $n^2 \log(n)$ calculations versus the conventional time domain back-projection algorithm with n^3 calculations. Analysis of the algorithm involved working in

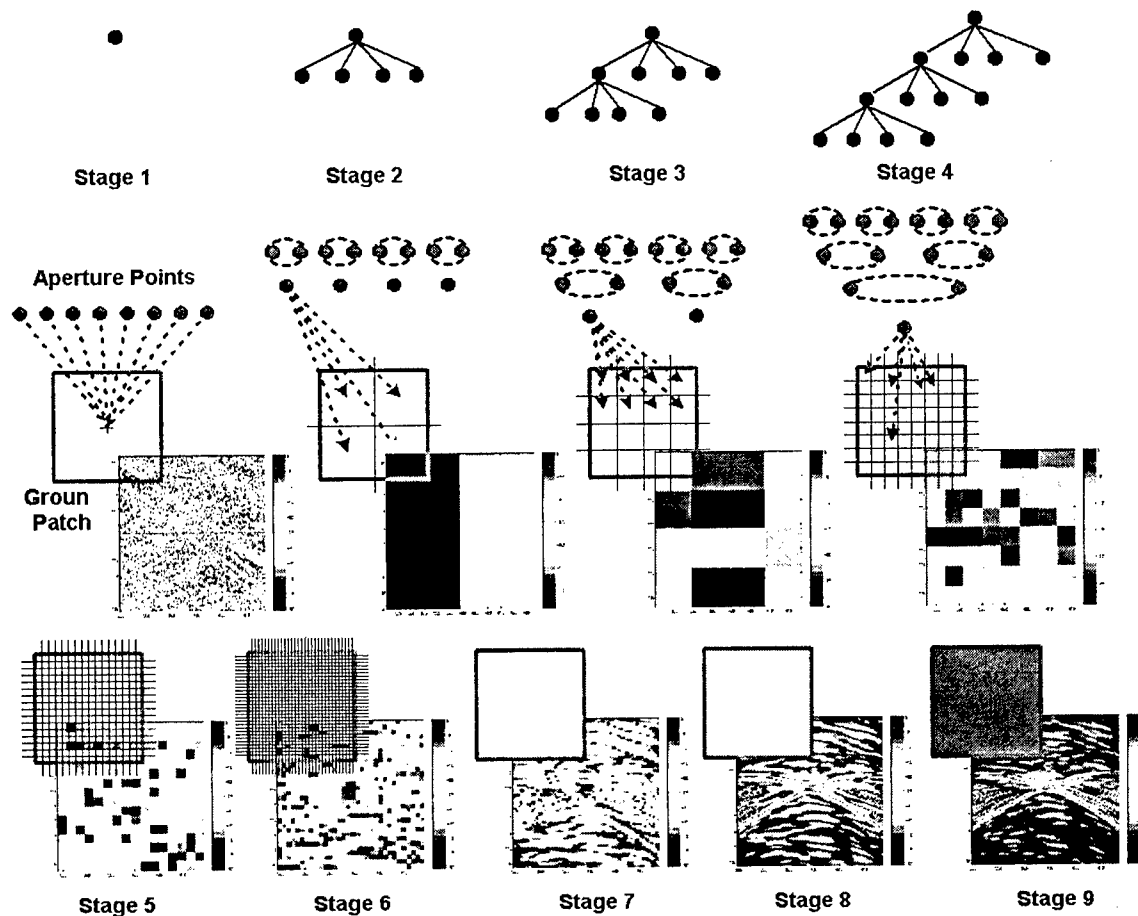


Figure 3.17.2: Quadtree Algorithm

conjunction with Georgia Tech, who had previously coded and run the Quadtree algorithm in Matlab. The Quadtree algorithm was run at Sanders on Matlab and modified to test the effects on the image using square root approximations. The algorithm analysis showed that the problem could be partitioned to separate aperture pairs that can be run in parallel. Sanders analyzed the algorithm dependencies and then mapped the algorithm for a single aperture pair to 2 CSRC devices.

The RCM may or may not be the ideal architecture for the scale recursive algorithm, but at a minimum it was felt that the RCM could be used to experiment with kernel pieces of the algorithm. Through experimentation, we will be able to determine if Adaptive Computing / CSRC can provide a significant improvement in performance for larger systems such as the DARPA FOPEN ATD platform.

The C code for the demonstration was decomposed into a mosaic manager part (running on PC host) and image former part (running on Power PC and/or CSRC) . The Mosaic Manager reads a chunk of raw radar data and sends it to the RCM, which runs the Quadtree image formation algorithm. The RCM forms the image and returns the frame to the Host PC. The Mosaic manager continues to send chunks of raw radar data until the entire image is formed and stitches the formed frames together. The plan was to host the Quadtree entirely in software on the PowerPC residing on the RCM, then migrate pieces

of the algorithm into the CSRC devices for acceleration. In this manner we could benchmark the speed up using the CSRC devices

The Quadtree algorithm has a lot of flow control as a sequence of loops (as seen in Figures 3.17.3 & 3.17.4), with the number of iterations changing at each stage of recursion. The main calculations are for the indexes that point to the data that is to be interpolated. Sanders identified the core sub-routine of the Quadtree to be the calculation of parent to child aperture distances. Sanders has completed all the software to implement the demonstration running on the PowerPC and have successfully demonstrated the implementation of the Quadtree algorithm on the RCM board utilizing the PPC750 as described above. In addition, Sanders has developed the GUI for the code that will display the image in an OpenGL 3D view. This GUI viewer of the Quadtree algorithm results after processing in the RCM (PPC750) with host mosaicing can be seen in Figure 3.17.5. Although all of the CSRC contexts required to migrate the “inner loops” to the CSRC devices (acting as coprocessors) were designed and simulated they were never implemented on the CSRC ICs due to funding shortcomings. However, the subsequent discussion is provided to explain the methodology and approach for mapping the Quadtree algorithm to the CSRC technology on the RCM board

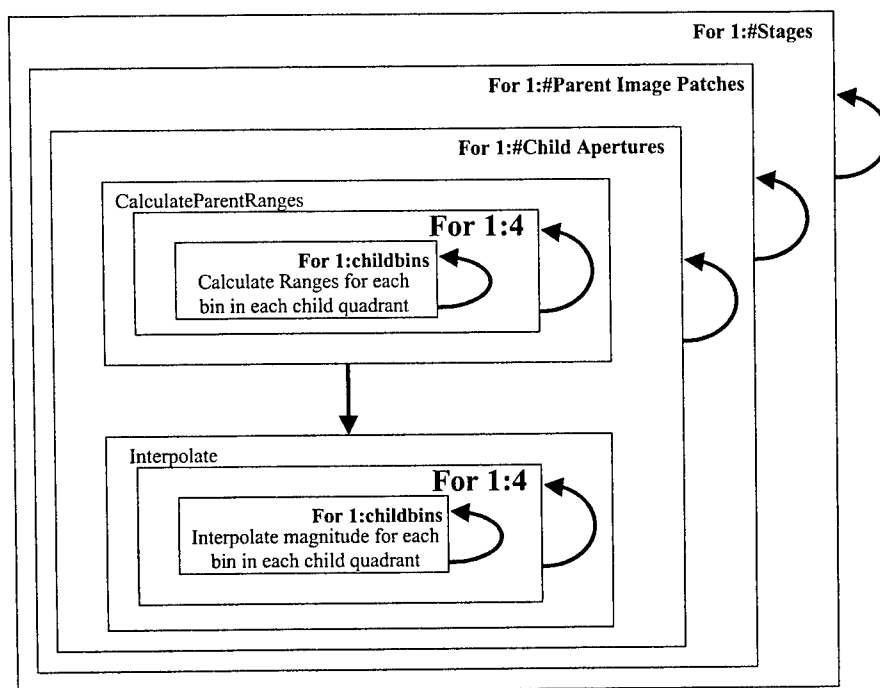


Figure 3.17.3: Iterative Loops of Quadtree Algorithm

The algorithm is structured in a way that one can partition the computation by aperture pairs. For example, with 2^M apertures and 2^N processors ($M \gg N$) each processor can be given 2^{M-N} aperture pairs to work on recursively without any data sharing between nodes. This would result in 2^N child apertures which could be processed with 2^L processors ($N > L$), again resulting in a partition of 2^{n-L} aperture pairs per processor and

2^L child apertures. It is interesting to note that as the recursion continues, the number of apertures decreases by 2 and the number of patches increases by 4.

Since the number of patches is increasing as the number of apertures is decreasing, each processor winds up doing more work in the index calculations and less interpolation.

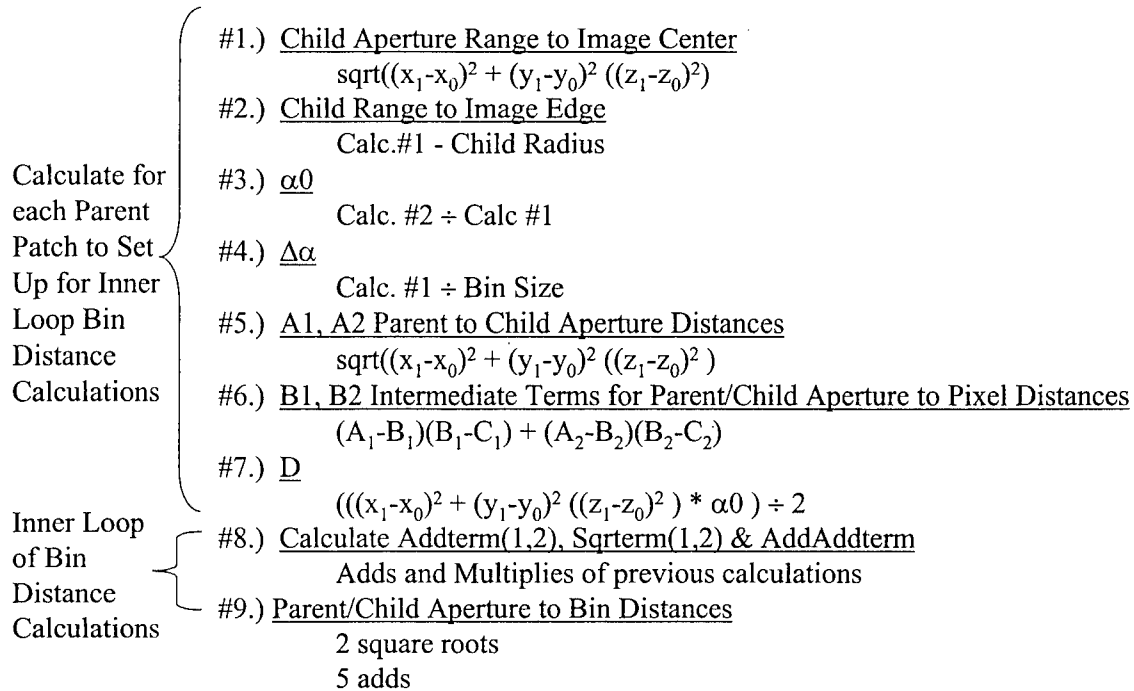


Figure 3.17.4: Iterative Loops of Quadtree Algorithm

The algorithm developers have observed this fact during simulation. Since we need to converge the apertures, eventually, into one processor, a bottleneck occurs. The first reaction is to find some way to adaptively begin partitioning the work by image patches. The problem here is that each image patch requires access to every aperture pair. Thus the memory or memory bandwidth requirement to be able to store or move all the apertures across the processing nodes would explode. This situation is compounded by the fact that the apertures created are larger than the ones in the previous recursions.

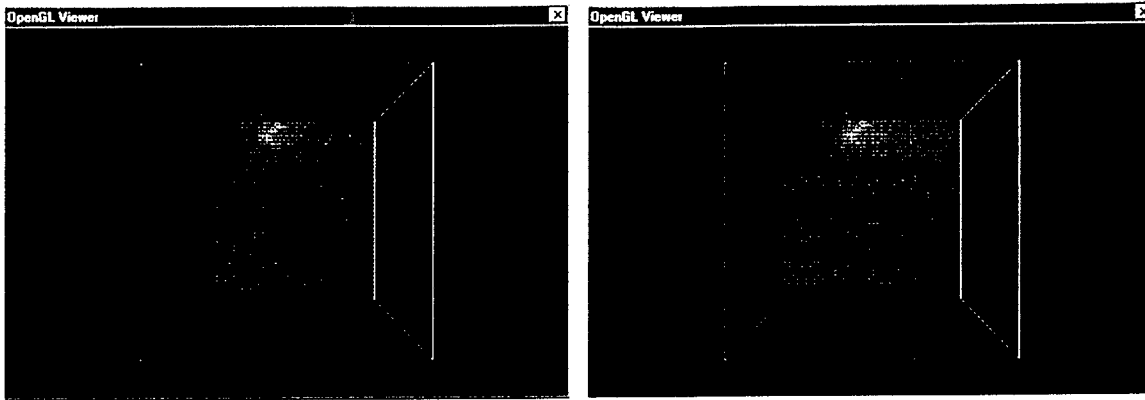


Image Formed Using *PC ONLY*

Image Formed Using *RCM*

Figure 3.17.5: OpenGL View of Wideband SAR Image Formation on RCM Utilizing PPC750 and host Mosaicing

Because of the complex control flow, the Quadtree algorithm does not lend itself well to being accelerated with current FPGA technology. However, it appears that the Quadtree algorithm would be a good candidate for Context Switching Reconfigurable Computing technology. As an example of how the CSRC might work, we envisioned a two-CSRC architecture being controlled by a general purpose processor (ie. the RCM board). The general purpose processor (GPP) (PPC750) would manage the stage of recursion, the number of iterations, and the interpolation. The GPP would load the CSRCs with desired parameters to compute the indexes and retrieve the indexes when calculated. A data dependency diagram was constructed and used to map the index calculations into the contexts of each of the CSRCs. The challenge is to balance operations so that data is available to both CSRCs so that neither of them sits idle.

To drive our architectural design we used the imaging geometry and rates for a countermeasure tactical UAV application. We will define real-time as the ability to image one ground patch for every synthetic aperture flown (with latency). Our goal is to demonstrate that the CSRC can accelerate the Quadtree algorithm by rapidly changing at the operator level and than that it can be used to adaptively switch to another algorithm to accelerate the overall application.

The Quadtree algorithm was modularized so that a part of it ran in the processor (PowerPC on RCM) and the rest in the CSRCs. In this case, the processor would manage the stage of recursion, the number of iterations and the interpolation. The processor would load the CSRCs with the desired parameters to compute the indexes and retrieve them when calculated. Since index calculation is more computational intensive, it has been considered for the CSRCs working in parallel with the processor.

The computations that are done in the CSRCs are: addition, multiplication, division, and square root. The first step was to settle upon the number of bits that will be used for the input and the output data, and also the number of bits that will be needed for the above computations so that we get a proper output from the Quadtree algorithm. The initial implementation of the quad tree algorithm has been done in MATLAB, hence, extensive

simulations were done to get the optimum number of bits for the part of the algorithm that will be shifted in the hardware (CSRCs). The next step was to determine the approximate algorithms to be used for the division and the square root.

Table 3.17.1: Bit Widths and Approximation Algorithm:

Input and Output data (for CSRC):	16 bits
Addition/Subtraction (parallel implementation):	16 bits
Multiplication :	17 X 17 bits (magnitude, by BRAUN Array multiplier), 6-bit exponent (signed), take the highest 16 bits of the multiplied magnitude as the output magnitude after exponent adjustment.
Division :	32-bit dividend with lower 16 bits as "0", 16 bits divisor, giving 16 bits quotient. Semi-parallel implementation.
Square Root :	Series Expansion Approximation has been used (for the first 8 terms).

Table 3.17.2: Determination of the Contents of each Context of the Two CSRCs:

CSRC I:

- Context 1: 3 differences in parallel, storage of 16-bit data, control to initiate Multiplier context, and storage registers (C, A1, A2, B1, B2, BinSize, AddAddTerm, Child Radius, Alpha0, D).
- Context 2 : 17 X 17 Parallel Braun Multiplier, Shifter, Accumulator, Exponent and mantissa adjustment blocks.
- Context 3: 2 adders for parallel addition, 16-bit registers to store AddAddTerm, SqrtTerm1, AddTerm1, $B1/(\text{Sqrt}(C))$, $D/(\text{Sqrt}(C))$.
- Context 4: Square Root Front End

CSRC II:

- Context 1: Square Root Front End
- Context 2: One Semi-Parallel Divider
- Context 3: 2 adders for parallel addition, 16-bit registers to store AddAddTerm, SqrtTerm2, AddTerm2, $B2/(\text{Sqrt}(C))$, and $D/(\text{Sqrt}(C))$.
- Context 4: 17 X 17 Braun Multiplier, Shifter and Accumulator. Exponent and mantissa Adjustment blocks.

Note that the Square Root and the Multiplier blocks have been replicated in each CSRC so that parallel computations are possible.

Finally, the algorithm flow as it was mapped to the CSRC devices on the RCM board is seen in Figure 3.17.6.

CSRC #1 @ 25MHz							CSRC #2 @ 25MHz							
Step	Context	Operation	#Iterations Option A	Option A Proc. Time	#Iterations Option B	Option B Proc. Time	Latency	Context	Operation	#Iterations Option A	Option A Proc. Time	#Iterations Option B	Option B Proc. Time	Latency
1		Compute 9 Differences	9	400	6	280	1							
2	2	Compute multiply	3	760	3	760	16							
		Compute sum of squares (2 adds)	1	80	1	80	1		Compute square root(range to center) and subtraction (child range to image edge) only calculated on first sum of squares calc.	1	40	1	40	
3									Compute divides (Alpha0 DelAlpha) only calculated on first	2	80	1	40	
		Jump to Step2	3	2520	4	3360								
4	2	Compute multiply for sqterms & addterms	1	680	1	680	16						0	
5	2	Calculate square for the addaddterm	1	680	1	680	16		Compute sums for squaterms and addterms	4	160	2	80	
6	2	Compute multiply for addaddterm	1	680	1	680	16							
7	2	Compute multiplies for sqterms & addterms	4	160	2	80	0							
8	1	Compute sums for sqterms	2	120	1	80	1							
9								4	Add addterms and sqterms	4	200	4	200	1
10	4	Store parent range to pixel distances to memory	1	200	1	200	4	1	Compute square roots for Parent range to pixels	2	1680	2	1680	40
		Jump to Step9	#Bins											
		Jump to Step1	4											

Figure 3.17.6: CSRC Mapping of Quadtree Algorithm Inner Loop

• 4.0 Conclusions

The Context Switching Reconfigurable Computing (CSRC) program was an overall success. The program focused on the development of a context switching device, design tools for these devices, design methodologies to exploit this technology, and new algorithmic techniques and associated tools to facilitate the widespread DoD and commercial application of this revolutionary device technology. The context switching device developed under the CSRC program is an FPGA that is capable of complete reconfiguration on a single clock edge, storing four configurations on chip, and allows sharing of data between "contexts" (or configurations). CSRC technology provides the signal processing and adaptive computing community with an additional dimension of flexibility, time. It is believed that CSRC technology will result in a performance improvement of two orders of magnitude over conventional DSP and FPGA implementations for many signal processing algorithms. This is by eliminating the need to continually move data on and off chip, instantiating application specific hardware when it is needed, and allowing for data-dependent hardware instantiation.

Development of the CSRC hardware proceeded in several stages. The first stage explored the CSRC concept, as well as guided the software tools' development effort, through the development and use of a high level behavioral model. The second stage emphasized the architect, design, fabrication, and test of a prototype logic device which supports context switching reconfigurable computing. Effort to develop new models of computation which are based on context switching reconfigurable logic and to develop software tools and models, facilitating algorithm, partitioning and mapping, as well as place and route (PPR) to foster exploitation of this revolutionary technology was undertaken.

The program has seen the successful development of the world's first context switchable FPGA, tools and methodologies for the development of applications utilizing the technology, and a computer module that employs the CSRC technology. It is Sanders' recommendation that this technology be placed in the hands of as many industry and ACS community researchers as possible so that the true potential of the CSRC technology can be reaped. It is obvious that the addition of two degrees of freedom to conventional FPGAs (time and "depth") truly deserves new paradigms. In fact, Sanders has seen that the CSRC technology is most effective with applications that utilize the technology as a virtual coprocessor, in multi-threaded applications, and in applications when the algorithm can be run through the data rather than the data through the algorithm. Note that in the last case, reductions in off-chip access can drastically increase performance, reduce power, and even eliminate the need for additional ICs such as external memory devices.

Finally, the CSRC team has developed what is believed to be a revolutionary technology. The past several years have seen numerous journal papers hypothesizing about what could be done if a device that was capable of context switching existed. Unfortunately, these papers have only been able to be theoretical to date since no such device existed. It is hoped that as a result of the availability of the CSRC technology, many of the hypotheses

can be proven or disproved, while stimulating a plethora of new research that was inconceivable with this revolutionary technology. Although the CSRC device is smaller in capacity than the state of the art FPGA devices, the device should be thought of as a proving ground for the technique and computing concept of context switching. Note that a larger capacity device with an alternate architecture can always be developed using the latest semiconductor process. In conclusion, Sanders has successfully developed the enabling technology to facilitate experimentation of context switching reconfigurable computing, the employment of CSRC technology in applications, and ultimately the deployment of runtime reconfigurable systems for DOD and commercial systems.

5.0 References

- [1] A. DeHon, "DPGA-Coupled Microprocessors: Commodity Ics for the 21st Century", IEEE Workshop on FPGAs for Custom Computing Machines, 1995.
- [2] A. DeHon, "Reconfigurable Architectures for General-Purpose Computing", PhD Dissertation –MIT, 1996.
- [3] R. Bittner & P. Athanas, "Wormhole Run-Time Reconfiguration", ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 1997.
- [4] J. Burns, A. Donlin, J. Hogg, S Singh, M. de Wit, "A Dynamic Reconfiguration Run-Time System", IEEE Symposium on FPGAs for Custom Computing Machines, 1997.
- [5] S. Kelem, "Mapping a Real-Time Video Algorithm to a Context-Switched FPGA", Poster Session, IEEE Symposium on FPGAs for Custom Computing Machines, 1997.
- [6] R. Ong, "Programmable Logic Device Which Stores More Than One Configuration and Means for Switching Configurations", US Patent 5,426,378, 1995.
- [7] S. Scalera, J. Murray, & S. Lease, "A Mathematical Benefit Analysis of Context Switching Reconfigurable Computing", Reconfigurable Architectures Workshop, 1998.
- [8] S. Trimberger, "A Time-Multiplexed FPGA", IEEE Symposium on FPGAs for Custom Computing Machines, 1997.
- [9] E. Tau, D. Chen, I. Eslick, J. Brown, A. DeHon, "A First Generation DPGA Implementation", FPD '94 – Third Canadian Workshop on Field-Programmable Devices, 1995.
- [10] J. Villasenor, B. Schoner, K. Chia, C. Zapata, "Configurable Computing Solutions for Automatic Target Recognition", IEEE Symposium on FPGAs for Custom Computing Machines, 1996.
- [11] M.J. Wirthlin & B.L. Hutchings, "Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration, IEEE Workshop on FPGAs for Custom Computing Machines, 1994.
- [12] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," *IEEE Transactions on Computer-Aided Design*, vol. 11, pp. pp. 4-15, 1993.
- [13] S. A. Cook, "The Complexity of Theorem-Proving Procedures," ACM Symposium on Theory of Computing, 1971.
- [14] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *Journal of the ACM*, vol. 7, pp. pp. 201-215, 1960.

- [15] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Transactions on Computers*, vol. C30, pp. pp. 215-222, 1981.

Appendix

1. The Design and Implementation of a Context Switching FPGA (FCCM98)
2. Efficiently Supporting Fault-Tolerance in FPGAs (FCCM98)
3. A Mathematical Benefit Analysis of Context Switching Reconfigurable Computing (RAW98)
4. Signature Hiding Techniques for Intellectual Property Protection (ICCAD98)
5. Watermarking Techniques for Intellectual Property Protection (DAC98)
6. Fingerprinting Digital Circuits on Programmable Hardware (Workshop on Information Hiding 1998)
7. Real-Time Image Formation Effort Using Quadtree Backprojection And Reconfigurable Processing(ARL/FEDLAB Symposium 1999)
8. Protecting Ownership Rights of a Lossless Image Coder Through Hierarchical Watermarking (IEEE Workshop on Signal Processing Systems 1998)
9. On-line Fault Detection for Programmable Logic (ARVLSI97)
10. A Case Study of Partially Evaluated Hardware Circuits: Key-Specific DES (FPL97)
11. FPGA Fingerprinting Techniques for Protecting Intellectual Property (CICC98)
12. Low Overhead Fault-Tolerant FPGA Systems
13. Evaluating the Benefits of Hardware Context Switching for Automatic Target Recognition
14. Data-Specific Number Factoring on Context Switching Configurable Computers
15. Suitability of Bin Packing for Context-Switching Configurable Computing Technology
16. Algorithms for Runtime Tolerance of Interconnect Faults on Diverse FPGA Architectures
17. Dynamic Circuit Generation for Solving Specific Problem Instances of Boolean Satisfiability
18. Georgia Tech presentation on preliminary CSRC mapping of UWB SAR

The Design and Implementation of a Context Switching FPGA

Sanders, A Lockheed Martin Company
PTP2-B007
65 River Road
Hudson, NH 03051

Stephen M. Scalera
sscalera@sanders.com

Phone (603) 885-0679 FAX (603) 885-7623

Jóse R. Vázquez
jvazquez@ede.sanders.lmco.com

Phone (603) 885-0746 FAX (603) 885-7623

Abstract

Dynamic reconfiguration of field programmable gate arrays (FPGAs) has recently emerged as the next step in reconfigurable computing. Sanders, A Lockheed Martin Company, is developing the enabling technology to exploit dynamic reconfiguration. The device being developed is capable of storing four configurations on-chip and switching between them on a clock cycle basis. Configurations can be loaded while other contexts are active. A powerful cross-context data sharing mechanism has been implemented. The current status of this work and future work are described.

1 Introduction

History has seen the methodologies of computing evolve from *fixed* hardware and *fixed* software (ENIAC), to *fixed* hardware and *reconfigurable* software (microprocessors), to *reconfigurable* hardware and *reconfigurable* software (FPGAs). FPGAs have traditionally been utilized in applications that demand the performance of application specific integrated circuits (ASICs) while maintaining the flexibility and rapid design cycle afforded by the use of digital signal processors (DSPs). Although FPGAs are not ideally suited for either requirement, they do offer an excellent compromise. In the recent past, many research efforts have examined the possibility of performance enhancement due to run-time reconfiguration. However, the best of today's commercially available technology requires milliseconds to reconfigure. This reconfiguration time, although acceptable for some applications, such as the *SPEAKEasy* reconfigurable "softradio" developed by Sanders, is an unacceptable delay for most real-time systems. Although partial reconfiguration can reduce the required reconfiguration time, this is believed to be an alternative approach to dynamic reconfiguration. Being able to *completely* reconfigure an FPGA at a rate that far exceeds the necessary persistence of a hardware function,

while being able to share data between configuration instantiations is believed to be tomorrow's reconfigurable computing computational model. This model of computation shall be referred to as *context switching reconfigurable computing* and is a natural extension of today's methodology. Arguably, context switching is not unlike the very first mode of computation. In essence, clock-cycle dynamic reconfiguration can be viewed as *fixed* hardware and *fixed* software since the available hardware can be thought of as being virtually infinite – *virtual hardware*.

The context switching reconfigurable computing (CSRC) technology being developed by Sanders extends commercially available field programmable gate array (FPGA) devices to include high speed changes between a number of programmed functions without the need for additional FPGAs. Each configuration, referred to as a *context*, in a CSRC FPGA has the functionality similar to that of many commercially available FPGAs. The context switching can occur at significantly higher speeds than the rate at which current FPGA technology can reconfigure. In addition, unlike commercial FPGAs, where reprogramming destroys any resident data, the CSRC FPGA affords the capability of data sharing between contexts.

The concept of virtual hardware is an obvious benefit of dynamic reconfiguration. If configurations can be swapped in and out of an FPGA upon demand at a real-time system rate, only the necessary hardware need be instantiated at any given time. In this manner, a virtually infinite algorithm cache or an infinite coprocessor can be conceived. In other words, a high level system scheduler can instantiate hardware as needed. In this manner, a reduction in size, weight, and power can be achieved. Additionally, given the CSRC FPGA, if the processing requirements specify a sequential application of

algorithms, the context layers can be set up to share data such that the output of one algorithm is immediately available as the input to the next algorithm upon a context switch. This is not possible with contemporary FPGAs.

A natural extension of the algorithm cache mode of computation is the concept of mission phase reprogrammability. As seen in Figure 1.1, an entire mission can be mapped to a CSRC device. In this case, different contexts can house different algorithmic phases of a mission without requiring that an algorithm be confined to a single context, depicted as layers in Figure 1.1.

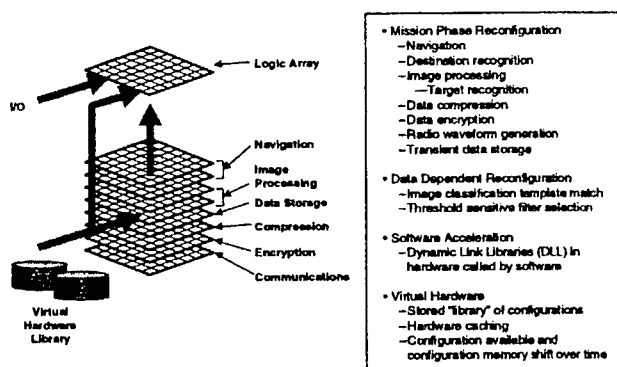


Figure 1.1: Reconfiguration Benefits

Although Figure 1.1 identifies the obvious modes of computation for gaining a performance enhancement, it is believed that the true potential of context switching requires a paradigm shift in algorithm implementation. The capabilities of the CSRC architecture, which extend dynamic reconfiguration to context switching, have the potential to provide improved implementations of signal processing algorithms over those currently available through commercial FPGAs. The inherent ability of CSRC to quickly perform different tasks and share results among different configurations allows one to approach algorithms from a different perspective, enabling mathematical implementations previously inconceivable without context switching.

Although the past few years have seen much interest in context switching reconfigurable computing, it is believed that this paper describes the first design and implementation of such a device. Up until now, all of the substantiated work on this model of computation has been theoretical. The emphasis of this paper is reveal the architecture of the context switchable FPGA being developed by Sanders. It is hoped that this paper will spark new ideas and facilitate algorithmic research that is targeted towards a specific and real architecture. With this achieved, as the world's first context switchable

silicon becomes available, members of the adaptive computing systems (ACS) community will be capable of taking full advantage of this new technology. IC development status is advanced in section 3.

2 Architecture

Experience has shown that FPGAs afford the greatest performance benefit when they are used to implement algorithms with deep pipelines. However, pure dataflow algorithms are rare. In fact, generating pipeline control signals, implementing state machines, and interfacing with external RAM or other integrated circuits, are critical, although not typically areas of performance enhancement, to an FPGA's successful system integration. With this in mind, the CSRC device was designed to be a 4 bit DSP dataflow engine that is simultaneously capable of efficiently implementing glue logic. However, since FPGA performance enhancements are oftentimes achieved by implementing the minimum required bitwidth, the CSRC device was developed to allow users to implement scalable pipelines such that the wordwidth can be of any size.

2.1 Data Pipes

The CSRC device is arranged into 16-bit wide data pipes. Each pipe is formed by a plurality of context switching logic arrays (CSLAs) as seen in Figure 2.1. A single CSLA is capable of processing two 16-bit words and outputting a 16-bit result. The result of a CSLA is available as an input to the two adjacent CSLAs in the pipe. Hence, a pipe can naturally be used as a data path. Information can easily flow from one end of the pipe to the other. It is important to point out that in this device data can non-preferentially flow in both directions. This feature has great utility when sharing data among different contexts. For example, one context could process data from left to right, storing it's final result in the right-most set of registers. Note that is quite possible that the final result of a single context is actually an intermediate result of the entire algorithm. Given this situation, an incoming context can pick up where its predecessor context left off by acquiring the intermediate data deposited on the rightmost portion of the pipeline and processing it in a pipeline that flows from right to left. From this simple example, it can be seen that a data path that does not favor data flow in either direction is more efficient for context switching hardware because it alleviates the need to reroute data from its physical origin in one context to its physical input in the subsequent context.

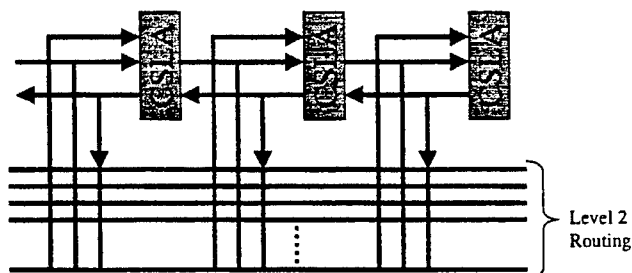


Figure 2.1: 16 Bit Data Pipe Comprised of CSLAs

Level 2 routing can be found alongside the pipe and consists of 16-bit busses. See Figure 2.1. These busses are not segmented and run the entire width of the CSRC device. This type of bussing scheme implies that a signal driven onto level 2 routing is available to any CSLA in the pipe. Additionally, this approach affords the possibility of faster and less complicated programming tools than segmented approaches because the timing is more deterministic. Each CSLA has two 16-bit inputs, each of which is capable of tapping into any of the Level 2 routing busses. Similarly, the CSLA's 16-bit output can drive any of the Level 2 routing busses. Note that Level 2 routing can be utilized as a bus architecture, can be broken down and utilized by individual bits, or can be employed as any combination of these.

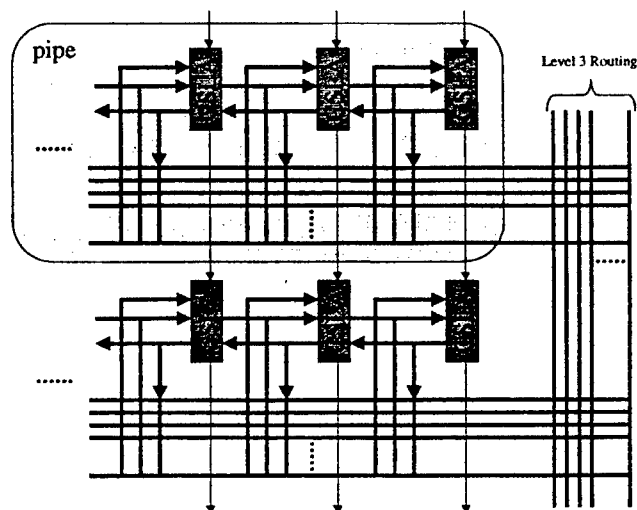


Figure 2.2: Level 3 Routing Bridges Pipes

The CSRC device is formed by stacking up pipes one on top of the other. Corresponding CSLAs on adjacent pipes have dedicated wiring that allows them to pass along their carry bit. This feature allows two adjacent pipes to be bundled together and be used as a single 32-bit wide data path. In actuality, physical 16-bit pipes can be broken down into smaller logical pipes. Although hardware is

optimized to break pipes into nibbles, pipes can be n-bits wide.

As seen in Figure 2.2, information driven onto a given pipe's Level 2 routing can be connected to Level 3 routing which in turn makes the data available to any Level 2 routing on the chip. Similar to the Level 2 routing structure, the Level 3 routing is not segmented and spans the device. Note that conceptually the Level 2 and Level 3 routing are perpendicular to each other.

I/O pins on the device are connected to Level 2 and Level 3 routing. All pins physically located on the top and bottom edges of the device connect to Level 3 routing. Pins on the left and right edges can connect to either Level 2 routing or directly into the dedicated routing that normally connects adjacent CSLAs.

2.2 Context Switching Logic Array

A single CSLA is primarily composed of 16 context switching logic cells (CSLCs) and Level 1 routing to interconnect them. Figures 2.3 and 2.4 depict a CSLA and the CSLA as it attaches to the Level 2 routing, respectively. The CSLCs are numbered 0 through 15 and their carry-in and carry-out chains are hardwired appropriately so they can function as a single cohesive unit. Level 1 routing consists of three 16-bit busses. Two of these 16-bit busses are inputs from the Level 2 routing. The third 16-bit bus is hardwired to the outputs of the CSLCs. Level 1 routing was designed with two modes of operation in mind.

2.3 Routing Modes of Operation

As previously mentioned, it is believed that the most beneficial FPGA is capable of exploiting its inherent DSP strengths while simultaneously being capable of implementing the often required glue logic. Hence, the CSRC FPGA has been designed with two modes of operation in mind: (1) Deep pipeline mathematical operations that can be of arbitrary bitwidth & (2) Random logic implementations that encompass control, state machines, and interfacing with external RAM or other integrated circuits. As a direct result, the CSRC FPGA exhibits two types, or modes, of routing.

2.3.1 Bus Routing

The first operational mode of routing is bus routing. The design goal was to provide users with the ability to route entire 16-bit words in and out of CSLAs while maintaining bitwidth order (i.e. the most significant bit (MSB) in the MSB position and the least significant bit (LSB) in the LSB position). Given that the four

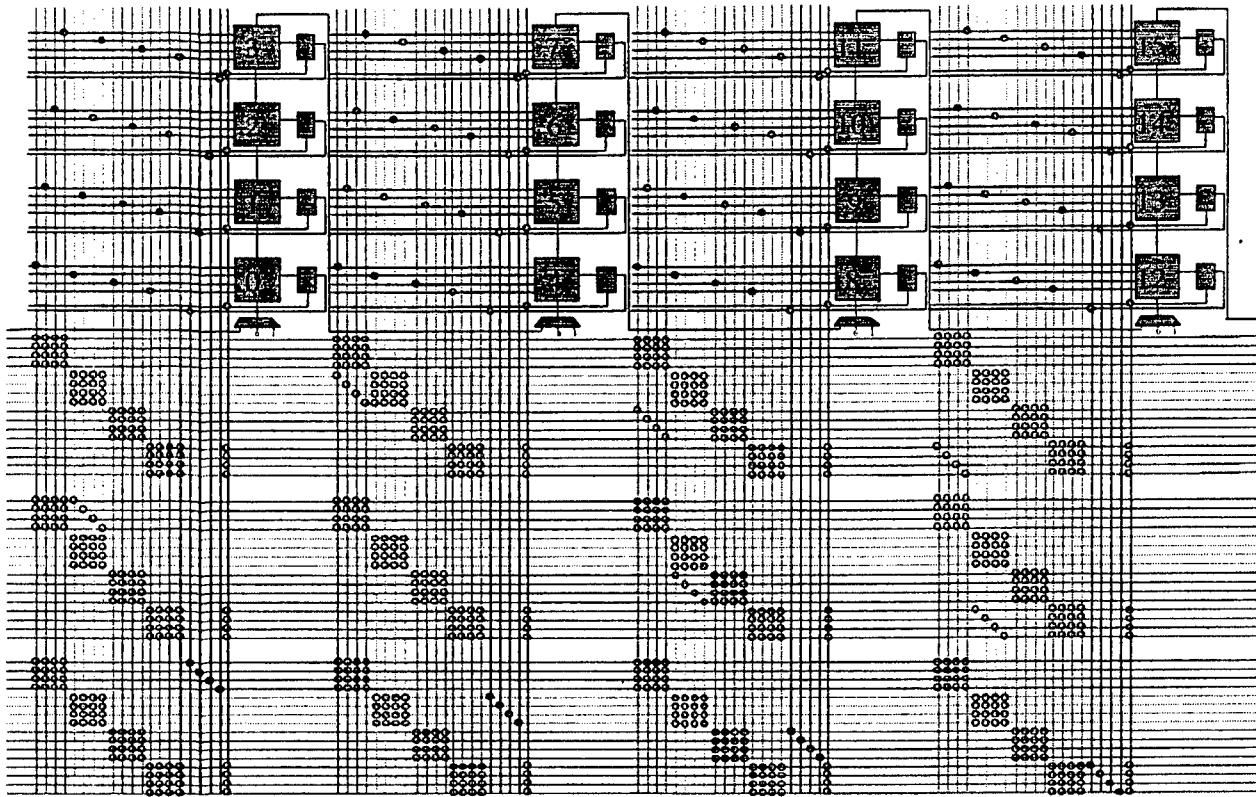


Figure 2.3: Context Switching Logic Array & Level 1 Routing

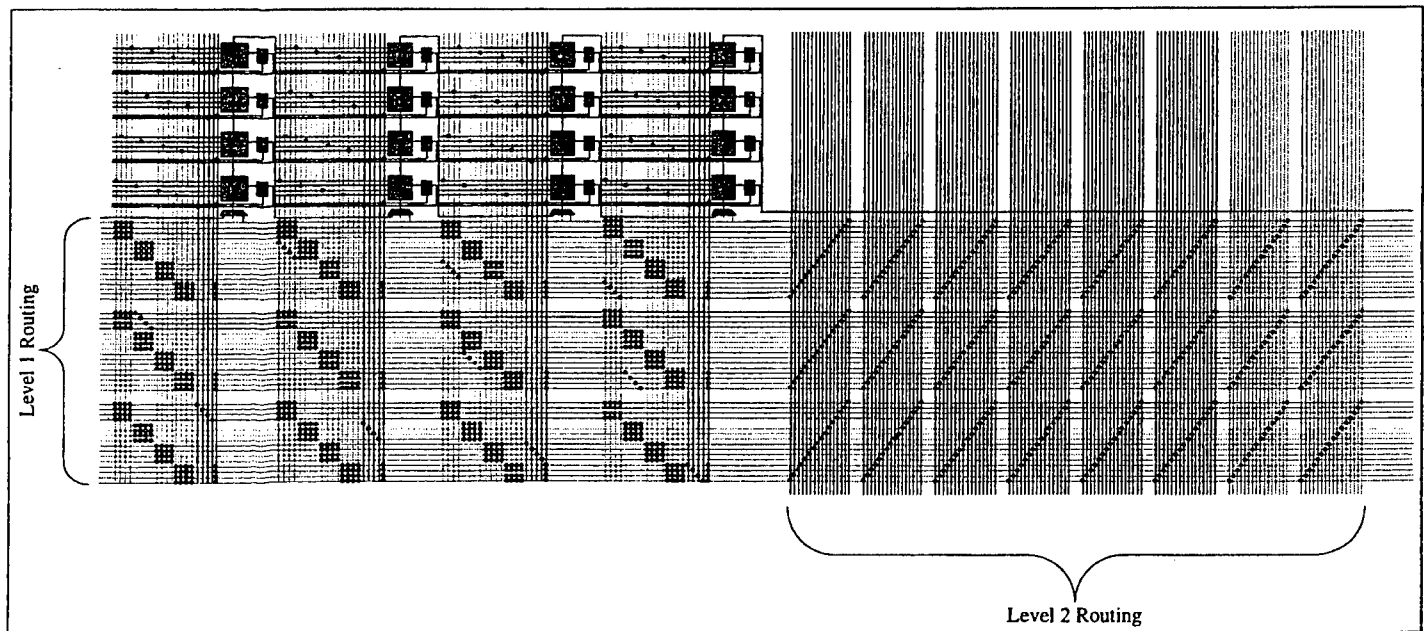


Figure 2.4: Context Switching Logic Array with Level 1 & Level 2 Routing

data inputs to the CSLC are labeled as A, B, C, and D, enough programmable connections are contained in the Level 1 switching matrices to ensure that one of the input buses can be routed into the A inputs of all of the 16 CSLCs. The least significant bit of the bus feeds the A-input of the least significant CSLC and so on. In essence, this bus can be considered the A-input (16-bits wide) for the entire CSLA under this bus routing mode of operation. Note that the second 16-bit bus can be used to feed the B inputs of the CSLCs within a CSLA in a similar fashion. The final bus connection is hardwired to the 16-bit output of the CSLA and attaches to the Level 2 routing. Note that this output is also a direct connect between neighboring CSLAs. As previously described, this non-directional direct connect allows for fast routing between CSLAs within a pipe by alleviating the need for Level 2 routing if the output of a pipe stage is feeding a neighboring CSLA.

2.3.2 Bitwise Routing

The second mode of operation is bitwise routing. No matter how data processing intensive a design might be there is almost always a need for control logic whether it is simple glue logic or more complex state machines. For this reason the bitwise routing mode of operation is necessary. The basic premise is that the output of any given CSLC within a CSLA should have at least one possible path to connect to at least one input of all other CSLCs within the same CSLA. A simple pattern of programmable connections was developed to enable this feature. All the A-inputs of all the CSLCs in a CSLA can tap into the four least significant bits of all three Level 1 routing busses (this includes the output bus to provide a means of local feedback without having to waste Level 2 routing resources). Similarly the B-inputs and the C-inputs tap into the next 4 bit bundles within each level 1 routing bus, and finally the D-inputs tap into the four most significant bits on every bus. As a result, the four least significant CSLCs, which drive the corresponding four-least significant bits of the output bus, are capable of driving any A-input on any CSLC within the same CSLA. For this reason these four CSLCs are known as "A-drivers" under the bitwise routing mode of operation. Similarly, B-drivers refers to CSLCs 4 through 7, C-drivers to CSLCs 8 through 11, and D-drivers to CSLCs 12 through 15. Furthermore, since connections between Level 1 and Level 2 and connections between Level 2 and Level 3 maintain proper bit order (LSBs to LSBs and MSBs to MSBs) any A-driver can drive the A-input of any CSLC anywhere in the chip. For these same reasons, the same functionality applies to the B, C, and D-drivers.

In addition to the four main inputs (A, B, C, & D), each CSLC has a clock enable / tri-state control line. Both of these control lines tap into the four most significant bits of

the three Level 1 buses, hence, they are controlled by D-drivers. As seen in Figure 2.5, the clock enable / tri-state control line is a single control line to the CSLC. For this reason, the user can choose to use this control line to control either the clock enable or the tri-state buffer.

2.4 CSLC

The CSLC is the heart of computation for the CSRC device. As seen in Figure 2.5, the CSLC is composed of carry logic, a four input lookup table (CSLUT), a context switching flip-flop (CSFF) and a tri-state buffer. The carry logic unit is capable of generating carry bits for either additions or subtractions. The carry logic chain is connected by dedicated connections. The chain can be connected, disconnected, or fed a logic zero or logic one every four bits. In this manner, the bus routing mode can be utilized to generate a pipeline granularity of four bits. However, in reality, the buswidths can be of an arbitrary bitwidth, n . Note that bitwidths with a modulo $4 = m$, where m is greater than zero, will disallow m CSLCs from supporting a mathematical pipe that requires the starting of a carry chain.

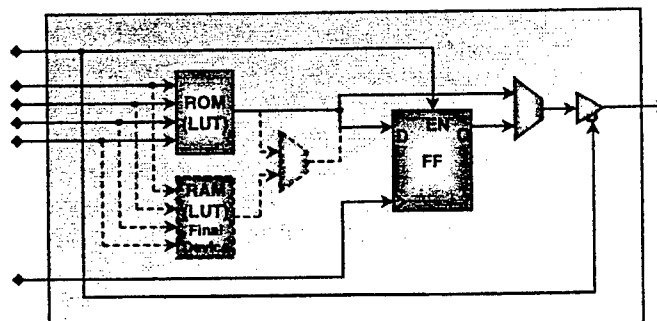


Figure 2.5: Context Switching Logic Cell Architecture

The outputs of the carry logic feed the CSLUT which consists of 16 context switching configuration bits (CSBits) that are multiplexed together. The 4 inputs serve as the select lines therefore implementing a programmable function. Note that the contents of each of the CSLUTs are unique in each context and specified in the configuration bitstream.

The CSBits implement context switching itself. Each CSBit holds a single programming bit for every context. However, only the active context's value drives whatever logic the CSBit is controlling. A detailed description of the context switching functionality afforded by these CSBits is described in section 2.6.

Unlike some commercial FPGAs, the lookup table can not serve as a memory element because the CSLUT is composed of CSBits, not SRAM. Instead a separate context switching RAM (CSRam) provides memory.

storage facilities. The CSRam, which is only available in the final CSRC device, implements the *global sharing scheme*. This data sharing scheme is similar to traditional blackboard data sharing. Any data written to a CSRam memory is available to all the CSRam elements that are physically collocated among different contexts. Whatever data value is last written into the active CSRam before deactivation of the current context will be seen by all other collocated CSRams upon the activation of their respective contexts. In fact, one can envision writing to a CSRam in one context and having its contents be used as a LUT in another context. Additionally, it is the CSRam that will allow for large amounts of data passing between contexts to facilitate modes of computation such as moving the algorithm through the data. This mode of computation is advantageous due to the fact that the on/off chip accesses are minimized by loading the data on chip and keeping it on chip until the entire algorithm has been run on the data.

Both the CSRam and the CSLUT will coexist in the final CSRC device and their outputs will be multiplexed together. The select line of this MUX is yet another control line to the CSLC and it is connected to Level 1 routing in the same fashion as the clock enable / tri-state control (driven by D-drivers). The output of this MUX can then be registered or passed directly out of the CSLC as seen in Figure 2.5. In either case, the user has the ability to tri-state the output of the CSLC. Note that if the data is to be registered, it will be done in the context switching flip-flop (CSFF). During regular operation within a single context, the CSFF appears to the users as a normal D-flip-flop (DFF). The DFF connects to the global clock and it is controlled by the clock enable input to the CSLC.

2.5 CSIO

The context switching input/output cell is used to facilitate on/off chip data accesses. As can be seen in Figure 2.6, the CSIO cell is bi-directional, can provide latched or direct outputs, and has a programmable pull-up resistor on the output. In addition, the CSIO cell can tri-state its output. Since on/off chip access time is oftentimes a limiting factor of FPGAs, a programmable drive strength capability has been included to insure maximum performance. The output drive strength can be selected to be 2mA, 6mA, or 12mA. Finally, the flip-flop in the CSIO cell utilizes a different sharing scheme than the flip-flop in the CSLC. Note that since it is believed that sharing data between contexts within a CSIO cell is unlikely to be a key feature, the global sharing scheme is implemented for the CSIO cell DFFs rather than the more complex sharing scheme that is implemented in the CSLC's DFF.

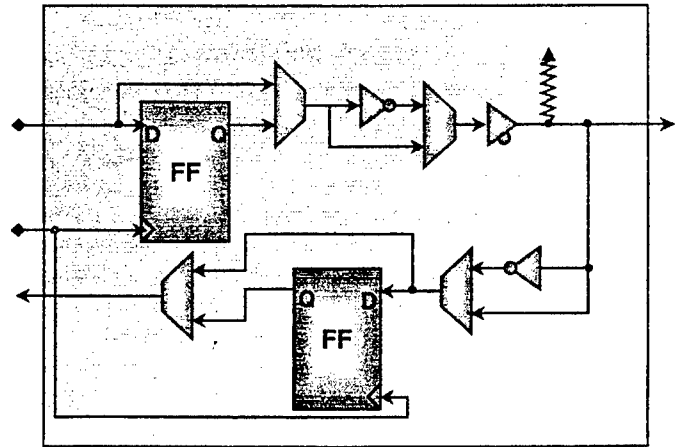


Figure 2.6: Context Switching Input/Output Cell Architecture

2.6 Data Sharing / Context Switching

Research indicates that the major benefits of context switching are afforded by sharing data between contexts and being able to switch between contexts very rapidly. For this reason, a great emphasis has been placed on the development of a device that meets both of these needs. The two sharing schemes that have been designed and implemented are Global Sharing and Private/Public Addressable Sharing (P/PASS). The global sharing scheme is used in the CSIO DFF and in the CSRam while P/PASS is used in the CSLCs by the CSFF. Global Sharing, as previously described, is simply a common memory element between all contexts. Hence, all contexts view these same memory elements and when any context writes to the memory element, the change is seen by all contexts upon their respective activation.

The data sharing scheme used by the CSFF, P/PASS, truly exposes the novelty of the CSFF and is depicted in Figure 2.7. With this type of sharing, each CSFF within each context supported in hardware has a corresponding register. These registers are known as *private* registers since they belong to a particular context and can only be accessed by a specific DFF within the context. Additionally, there is a single active register per CSFF. The active register is what the user actually utilizes during uninterrupted context execution. Upon switching contexts, the outgoing (active) context saves its intermediate values to its private registers. This feature enables many of the capabilities that the NSA would need to develop secure kernels by isolating intermediate data. Additionally, a context can *choose* to write its values to a *public* register (on a Logic Cell by Logic Cell basis) which can be addressed by any and all of the contexts. In this manner, the sharing of data between DFFs within contexts is enabled. The number of public registers available in a P/PASS implementation is independent of the number of contexts supported directly by hardware.

Hence, public registers must be addressed when used. Note that the CSRC device affords two public registers. Upon activation, a context can choose to restore its previous state by reading from the private register or it can opt to load a state from either public register (on a Logic Cell by Logic Cell basis).

P/PASS provides a means to keep secure data isolated while at the same time allowing data to be shared (if so desired) using public registers. This architecture scales to implementations with more contexts than hardware supports, allows sharing data between contexts that do not necessarily follow one another in time, and provides a clean and solid foundation to add features such as interrupt handling and hardware recursion.

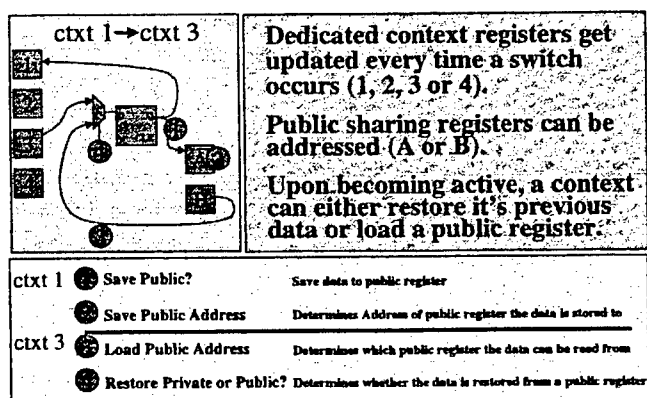


Figure 2.7: Private/Public Addressable Sharing Scheme

Since some modes of computation, such as the virtual coprocessor, require rapid reconfiguration, the CSRC device was designed to be capable of switching contexts on a single clock cycle. A key point to be made is that this single cycle context switching not only includes completely reconfiguring the CSRC device but completely executing all of the data sharing schemes. In fact, the active context can be swapped so rapidly that a context can be processing data on one clock edge, switch to a new configuration (including data sharing) and be processing data in the new configuration on the very next clock edge. SPICE simulations indicate that it is possible to switch contexts in fewer than 5 nanoseconds. A caveat to this rapid context switching is that time will be required to distribute the "switch to" lines throughout the chip. These lines indicate which context the device is supposed to switch to upon receiving the "switch" signal. However, given that the "switch to" lines are stable, the context switch can take place as described above. Note that this delay in switching is merely a latency and can therefore be factored into the logic that initiates a switch. Since the switch can be initiated by the active context or via external stimulus this latency is easily accounted for.

2.7 Programming

The bitstreams for the CSRC FPGAs are downloaded serially. The user is required to specify which context is about to be downloaded and then supply a clock and data. By repeating this process four times, the user can configure all four on-chip configurations. Note that the configuration being downloaded can not be active during configuration download. However, inactive contexts can be downloaded while another context is active and running. Additionally, a bitstream may be downloaded by the active context. In this manner, one can envision the possibility of passing compressed or encrypted bitstreams into the active context so that it may download an inactive context after uncompressing or decrypting the bitstream.

The device will power up, prior to downloading bitstream(s), in a known state possessed by all four configurations. This provides the user with the ability to determine if the device is operational prior to use. The known state, although yet undetermined will either be benign or of some simple functionality providing some level of built-in self-test (BIST).

Software will be available with the device that generates bitstreams for the CSRC devices from VHDL. The user will be required to manually partition the logic among the contexts if the algorithm should span multiple contexts. However, in such a case, the toolkit will be capable of simultaneously placing and routing the logic in each context so that intermediate data passage is optimized. This is believed to be a key element of context switching data sharing because if each of the contexts, which are to share data, are individually and sequentially placed and routed "in a vacuum", the placement restraints imposed on each context would compound and rapidly become prohibitively constraining.

3 Status

Sanders is developing this technology in two phases. The first phase involves the development of a small prototype version of the CSRC technology. This chip will be one of the world's first context switchable FPGAs and will serve both as a concept validation tool and a platform for acquiring empirical data about the performance enhancements afforded by this new technology. The subsequent phase entails the development and fabrication of a larger version of the prototype with a few additional features. Both the prototype and the larger more capable final device are full custom IC designs being designed and fabricated on National Semiconductor's .35μ line. The prototype CSRC device is being sent to fabrication in late February 1998 while the final device is scheduled to be fabricated in September of 1998.

4 Acknowledgments

This material is based upon work supported by DARPA/ITO under contract number F30602-96-C-0350 and Sanders IR&D.

5 References

- [1] A. DeHon, "DPGA-Coupled Microprocessors: Commodity Ics for the 21st Century", IEEE Workshop on FPGAs for Custom Computing Machines, 1995.
- [2] A. DeHon, "Reconfigurable Architectures for General-Purpose Computing", PhD Dissertation – MIT, 1996.
- [3] R. Bittner & P. Athanas, "Wormhole Run-Time Reconfiguration", ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 1997.
- [4] J. Burns, A. Donlin, J. Hogg, S Singh, M. de Wit, "A Dynamic Reconfiguration Run-Time System", IEEE Symposium on FPGAs for Custom Computing Machines, 1997.
- [5] S. Kelem, "Mapping a Real-Time Video Algorithm to a Context-Switched FPGA", Poster Session, IEEE Symposium on FPGAs for Custom Computing Machines, 1997.
- [6] R. Ong, "Programmable Logic Device Which Stores More Than One Configuration and Means for Switching Configurations", US Patent 5,426,378, 1995.
- [7] S. Scalera, J. Murray, & S. Lease, "A Mathematical Benefit Analysis of Context Switching Reconfigurable Computing", Reconfigurable Architectures Workshop, 1998.
- [8] S. Trimberger, "A Time-Multiplexed FPGA", IEEE Symposium on FPGAs for Custom Computing Machines, 1997.
- [9] E. Tau, D. Chen, I. Eslick, J. Brown, A. DeHon, "A First Generation DPGA Implementation", FPD '94 – Third Canadian Workshop on Field-Programmable Devices, 1995.
- [10] J. Villasenor, B. Schoner, K. Chia, C. Zapata, "Configurable Computing Solutions for Automatic Target Recognition", IEEE Symposium on FPGAs for Custom Computing Machines, 1996.
- [11] M.J. Wirthlin & B.L. Hutchings, "Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration, IEEE Workshop on FPGAs for Custom Computing Machines, 1994.

Efficiently Supporting Fault-Tolerance in FPGAs

John Lach

UCLA EE Department
56-125B Engineering IV
Los Angeles, CA 90095
(310) 794-1630

jlach@icsl.ucla.edu

William H. Mangione-Smith

UCLA EE Department
56-125B Engineering IV
Los Angeles, CA 90095
(310) 206-4195

billms@icsl.ucla.edu

Miodrag Potkonjak

UCLA CS Department
4532K Boelter Hall
Los Angeles, CA 90095
(310) 825-0790

miodrag@cs.ucla.edu

1. ABSTRACT

While system reliability is conventionally achieved through component replication, we have developed a fault-tolerance approach for FPGA-based systems that comes at a reduced cost in terms of design time, volume, and weight. We partition the physical design into a set of tiles. In response to a component failure, we capitalize on the unique reconfiguration capabilities of FPGAs and replace the affected tile with a functionally equivalent tile that does not rely on the faulty component. Unlike fixed structure fault-tolerance techniques for ASICs and microprocessors, this approach allows a single physical component to provide redundant backup for several types of components. Experimental results conducted on a subset of the MCNC benchmarks demonstrate a high level of reliability with low timing and hardware overhead.

1.1 Keywords

FPGA, fault-tolerance

2. INTRODUCTION

2.1 Motivation

While once FPGAs were mostly applied to prototyping, logic emulation systems and extremely low volume

applications, they now are used in a number of high volume consumer devices. FPGAs are also now being used in more exotic applications. For example, the Mars Pathfinder mission launched in 1996 by NASA relies on Actel FPGAs for some system services. Unlike early applications, these high volume and mission critical systems tend to have stringent reliability requirements [19]. Thus, there is a drive from the user community to improve reliability through some level of fault-tolerance.

Unfortunately, current technology trends tend to make FPGAs less reliable. FPGA vendors have been moving down the same path of smaller device size as the rest of the semiconductor industry. Electronic current density in metal traces will increase as device feature size shrinks from 0.5 μm to 0.35 μm and smaller, which results in a greater threat of electromigration. As transistors shrink, the amount of charge required to turn them on reduces, which also makes the components more susceptible to gamma particle radiation. At the same time, FPGA vendors are moving to larger and larger dies in order to deliver more logic gates to their customers. The larger dies introduce more opportunities for failure and bigger targets for gamma particles.

Engineers traditionally respond to these threats through redundancy, such as replicating components (e.g. microprocessors and ASICs) or replicating logic internal to a component (e.g. Built-In Self-Repair (BISR)). However, replication is a particularly unattractive approach for FPGA systems given the common customer complaint that devices cost too much and do not provide enough equivalent logic gates. A better approach is to leverage the flexible nature of FPGA devices to provide replication at a much finer level. Conceptually, if a single logic block fails, it is often possible to find an alternate circuit mapping that avoids the fault. Most vendor place and route tools provide an option for reserving resources, and in the face of a fault, the tool could be invoked to search for a new placement which only uses functional components. The resulting system could provide reliability with very low overhead (i.e. by reserving only a few percent of the resources as spares for fault recovery). Unfortunately, this approach results in significant system downtime. Thus, the technique will not be sufficient for mission-critical applications with hard real-

Presented at FPGA'98

Monterey CA, February 22-24, 1998

time constraints. This approach also requires that the end user have the vendor place and route tools, which is usually not possible. It seems unlikely that the end consumer will wish to even know about an embedded FPGA, let alone worry about generating a new configuration for one. Finally, because each fault is distinct, each component would possibly require a unique circuit placement. These three factors combine to make the approach impractical.

We instead propose a technique for increasing FPGA system reliability with very low overhead. The target architecture for demonstration is a Xilinx 4000EX part, which is composed of an array of configurable logic blocks (CLBs). Nonetheless, we believe that this technique is applicable to a wide range of FPGA architectures. The place-and-route CAD tool maps a circuit net-list onto the array of CLBs and interconnect components. We propose partitioning the physical design into a set of tiles. Each tile is composed of a set of physical resources (i.e. CLBs and interconnect), an interface specification which denotes the connectivity to neighboring tiles, and a net-list. Reliability is achieved by providing multiple configurations of each tile. Furthermore, by using locked tile interfaces, the effects of swapping a tile configuration do not propagate to other tiles, thus reducing the storage overhead.

2.2 Motivational Example

Consider the Boolean function $Y=(A \wedge B) \wedge (C \vee D)$, which might be implemented in a tile containing four CLBs as shown in Figure 1. This configuration contains one spare CLB, which is available if a fault should be detected in one of the occupied CLBs. Upon detecting such a fault, an alternate configuration of the tile is activated which does not rely on the faulty CLB. Each implementation is interchangeable with the original, as the interface between the tile and the surrounding areas of the design is fixed and the tile's function remains unchanged. The timing of the circuit may vary, however, due to the changes in routing.

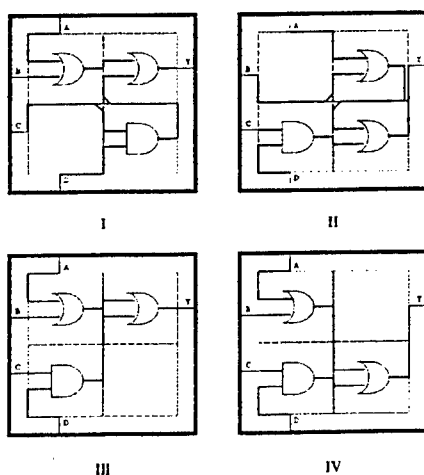


Figure 1: Motivational Example

This approach has three main benefits compared to redundancy-based fault-tolerance: very low overhead, the option for runtime management, and complete flexibility. The overhead required to implement this fine-grained approach, which can be measured in both physical resources on the FPGA (CLBs, I/O blocks, and routing) and timing, is extremely low compared to redundancy. Runtime management can be a very valuable feature of a system, particularly for mission-critical applications. This fault-tolerance approach handles runtime problems on-line, minimizing the amount of system downtime and eliminating the need for outside intervention. The flexibility that this approach provides allows for application specific solutions. The degree of fault-tolerance can be changed based on timing constraints, resource limitations, or presumed CLB reliability.

2.3 Paper Organization

The following two sections discuss background information, approach restrictions, and work related to fault-tolerance and FPGAs. Section 5 describes the details of the approach and implementation, and Section 6 introduces the formulas used to calculate the data for the experimental results in Section 7. Section 8 discusses future work on this topic, and the paper closes with some remarks summarizing the benefits of this fault-tolerance approach.

3. Preliminaries

In this section, we survey the relevant background material for the proposed approach. We present the targeted FPGA architecture, the fault model assumed, and techniques envisioned for supporting the testing and fault diagnosis steps of the approach.

3.1 FPGA Architecture Model

The new fault-tolerance approach is demonstrated using the Xilinx XC4000EX family as the target architecture, specifically the XC4028EXBG352 [24]. However, neither the general concept nor the optimization algorithms are specific to the 4000EX family, or even Xilinx architectures. Any FPGA architecture supporting the ability to reconfigure a large number of times could be used, such as the Altera 10k and the GateField flash memory devices. The approach is not applicable to anti-fuse systems, such as the Actel architecture, as they can not be reprogrammed.

3.2 Fault Model, Testing and Diagnosis

The proposed approach requires fault detection and a diagnosis method as a preprocessing step. We assume a widely used single stuck at, open, or short fault model [1]. It is interesting to note that our strategy actually covers many simultaneous faults as long as each tile (see Sections 5 and 6) has at most one faulty CLB. In its current form, our approach does not address interconnect faults. Note that

for local interconnects, interconnect faults will be expressed as a fault of the CLB to which it connects.

A number of schemes have been developed for detecting faults in FPGAs through exhaustive testing of the device architecture. Most of these approaches can be classified as off-line. For example, with Built-In Self-Test (BIST) [13, 21, 8, 3], the FPGA is loaded with a small testing circuit that is restricted to a specific physical region of the device, which is then used to test another portion of the device. The test circuit is moved across the device in a systematic manner until the entire device is thoroughly tested. The downside of these approaches is that they require the device to be taken off-line, which may not be practical in highly fault-sensitive, mission-critical applications. Fault-detection latency also increases as a result of an off-line approach. Recently, an on-line testing scheme has been developed for bus-based FPGAs that avoids these problems and may be well suited for fault-detection within this fault-tolerance approach [18].

4. Related Work

Related work can be traced along the following three lines of research: FPGA synthesis, fault-tolerant design, and FPGA yield enhancement.

A number of different FPGA architectures and synthesis techniques have been proposed and demonstrated [16, 2]. Conceptually, our fault-tolerance approach is closest to BISR techniques. The main targets for BISR are systems that are bit-, byte-, or digit- sliced. These types of systems include SRAM and DRAM memories [14], as well as systems designed using a set of bit planes and arithmetic-logic units (ALUs), assembled from ALU byte slices [19]. By far the most important use of bit-sliced BISR is in SRAM and DRAM circuits [17, 9, 22]. The bit-sliced BISR in memories significantly increases memory production profitability and is regularly used in essentially all modern DRAM designs. Among other BISR bit-sliced devices, the most popular and well addressed, from both a theoretical and practical point of view, are programmable logic arrays PLAs [4, 10, 23, 6]. A simple, yet powerful methodology for the implementation of ALU byte slices was proposed by Levitt et. al. [11].

Howard et. al. [7] and Dutt et. al. [5] have proposed using similar regularly structured BISR techniques for improving FPGA yield. Spare resources are allocated, and a manufacturing step is used to swap spare CLBs for faulty components. Altera uses this approach, along with on-chip fuses, to increase production yield on the 10K parts. Mathur and Liu have proposed using modified place-and-route tools to reroute part of the net-list in the vicinity of a faulty CLB [12].

Our approach is completely transparent to the existing CAD tool chain and exists as an intermediate step that is used in conjunction with existing synthesis and place-and-route

tools. Unlike the BISR techniques used in manufacturing, we are able to dynamically tolerate faults in the field. Finally, unlike Mathur and Liu, we are able to make timing guarantees (which is critical for real-time systems), require less system downtime, and do not require the end user to have access to FPGA CAD tools.

5. Approach

The key element of our approach to fault-tolerance is partially reconfiguring the FPGA to an alternate configuration in response to a fault. If the new configuration implements the same function as the original, while avoiding the faulty hardware block, the system can be restarted. The challenging step is to identify an alternate configuration efficiently. In this section, we elaborate on the key elements of our approach.

5.1 Tiles and Atomic Fault-Tolerant Blocks

We reduce the amount of configuration memory required by reducing the size of the component that is reconfigured. This is enabled by logically partitioning a design in a way that components can be independently reconfigured without impacting the rest of the design. In comparison with other alternatives, this approach also reduces the down time for devices that support partial reconfiguration and, more importantly, significantly increases the level of fault-tolerance with only nominal hardware and timing overhead. The key concepts for implementing the new approach are tiles and atomic fault-tolerant blocks (AFTBs).

Definition 1: A *tile* is composed of three elements: a set of CLBs and interconnect resources, a net-list which must be placed on those CLBs and routed across the interconnect, and a specification of how to interface the tile to adjacent tiles.

Definition 2: An *atomic fault-tolerant block* is one instance of a tile and has at least one spare CLB that serves to "cover" the faulty CLB(s).

Because each tile is associated with both physical resources and portions of the complete net-list, the design can only be partitioned into tiles after the complete net-list has gone through place-and-route once. By fixing the interfaces between the tiles (a constraint imposed on the place-and-route software), we create the opportunity to produce multiple partial configurations that satisfy the functional specification for a given tile, independently from the remainder of the design. Fault-tolerance is achieved by introducing spare resources into each AFTB so that, once a fault in a particular CLB is detected, a configuration of the tile's functionality (i.e. an AFTB) that does not utilize the faulty CLB can be activated.

Each tile has a set of AFTBs. An AFTB is independent from all AFTBs associated with other tiles by virtue of the fixed tile interface. Thus, selecting one AFTB for each tile

can assemble a complete configuration, under the condition that none of the AFTBs rely on a faulty component.

Tiling provides many advantages in the implementation of fault-tolerant FPGA systems. First, the amount of memory needed to store the set of AFTBs is smaller than the amount required to store a set of complete configurations. For example, consider a design that must be able to tolerate any single CLB fault and which maps into a 6x6 CLB array. It may be possible to divide the design into four 3x3 tiles (Figure 2). Assuming that one configuration of the complete 6x6 design requires X bytes of memory, the non-tiled approach would require $36 \times X$ of memory for fault-tolerance: one configuration for each CLB that is at risk. With our method, each tile would require 9 AFTBs. However, since each tile ($X/4$ storage bytes) is independent, the entire storage is only $9 \times X$, a 75% reduction from the non-tiled approach.

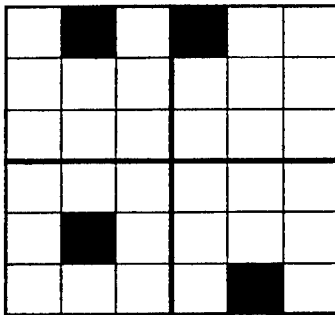


Figure 2: A 6x6 CLB design partitioned into 4 3x3 tiles
Tiling also increases reliability. For this example, the non-tiled approach could tolerate only one faulty CLB in the entire device. The tiled approach, however, is capable of tolerating any single fault in a tile but up to 4 faulty CLBs in the entire device.

The cost of increased fault-tolerance and reduced configuration memory is the possible introduction of more spare resources. For this example, the non-tiled approach reserves 2.7% of the CLBs to protect against a single fault, while the tiled approach reserves 11%. However, tiling opens up the opportunity to explore a rich design space. By choosing the tile size and amount of spare resources that are appropriate for system requirements, tiling provides a powerful tool to the designer.

One remaining issue involves circuit timing. While timing analysis tools are not completely reliable for FPGA devices, the Xilinx XACT Step software has proven to be reasonably accurate. We use this tool to determine the timing estimate of the initial configuration before tiling. Furthermore, because each individual AFTB is generated as a modification to the original configuration we have good timing estimates for any single failure. However, it is difficult to know what the circuit timing will be once multiple AFTBs have been activated in response to failures in more than one tile. In particular, the critical path for the

entire FPGA could pass through several AFTBs without falling on the longest path within any of them.

5.2 Synthesis Methods

The synthesis approach is organized in an iterative top-down manner. We start with a non-fault-tolerant base design and recursively partition it into tiles and AFTBs. We next check the feasibility of all fault scenarios in decreasing estimated level of difficulty. The idea is to terminate as early as possible those base designs that will not result in a feasible solution. We also calculate early the final reliability figures, so that the designer has an option of terminating the current search and starting one that has a higher reliability potential.

The synthesis is summarized in the following pseudo-code:

```

1.  while (!(complete || design possibilities exhausted)) {
2.      create initial non_ft_design;
3.      extract timing and area information;
4.      calculate design reliability;
5.      while (!(complete || tiling possibilities exhausted)) {
6.          partition design into tiles;
7.          if (!meet area criteria) break;
8.          while (!(complete || AFTB possibilities exhausted)) {
9.              partition tiles into AFTBs;
10.             calculate AFTB reliability;
11.             if (!meet reliability criteria) break;
12.             order tiles by ft realization difficulty;
13.             order AFTBs by ft realization difficulty;
14.             for (j=1; j<=# of tiles; decreasing difficulty) {
15.                 for (i=1; i<=# of AFTBs; decreasing difficulty) {
16.                     create ft_design(i,j);
17.                     if (!(success && meet timing criteria)) break;
18.                 } } } }

```

Lines 2-4 initialize the synthesis process for one instance of the base design. The place and route tool creates the base non-fault-tolerant design, and the relevant design characteristics are recorded. The procedure for the calculation of reliability is explained in Section 6.

Line 5 starts the synthesis algorithm and dictates that the loop will terminate upon the creation of a fault-tolerant design that meets all of the user specifications, including overhead (area and timing), level of fault-tolerance, and available memory. The loop will also terminate if the algorithm reaches the end of its exhaustive tile partitioning search, thus revealing that the specifications cannot be met for the given FPGA architecture and design generated in

line 2 In this case, the complete algorithm will be repeated using a different base design with increased spare resources throughout the FPGA

Line 6 partitions the design into tiles, as described in Section 5.1 The placement and shape of the tiles are determined by the following three key factors listed in decreasing order of importance: amount of interconnect across the tile interface, tile logic density, and tile size Our reliability calculation (see Section 6) indicates that large tiles result in higher reliability If the tiling attempt does not meet the user area specifications, the algorithm returns to the beginning of the tile partitioning loop for another tiling attempt Hard macros (and fast carry chains) also affect the placement and shape of tiles, as efforts are made to keep macros intact

Line 8 begins the AFTB partitioning algorithm, which terminates upon the successful creation of a fault-tolerant design meeting all user specifications or upon the exhaustion of all AFTB partitioning possibilities If the latter occurs, the algorithm returns to line 5 for re-tiling

Line 9 lays out the various AFTBs within a tile The number of AFTBs for a given tile depends on the desired level of fault-tolerance, the number of free CLBs in the tile, and the malleability and density of the logic The criteria used for partitioning the design into tiles are also used for this AFTB partitioning

Line 10 insures that the tile and AFTB partitions meet the user reliability specifications, and line 11 returns the algorithm to the beginning of the AFTB partitioning loop if they are not met If upon such a return the AFTB partitioning possibilities have been exhausted, the design must be re-tiled, returning the algorithm to line 5

Lines 12 and 13 facilitate early synthesis process failure detection Tiles and AFTBs that are less likely to successfully place and route should be attempted first, thus efficiently returning the algorithm to the beginning of the loop if a fault-tolerant design meeting user specifications is not possible with the current tile and/or AFTB partitioning The tile and AFTB characteristics causing them to be more difficult to realize include the criteria used in tile and AFTB partitioning The presence of macros, and therefore reduced logic malleability, may also impact the assigned order of realization difficulty

Lines 14 and 15 enforce the order defined by the two previous steps, as line 16 attempts to configure the various tiles and AFTBs If the design can not be configured or if the configuration doesn't meet user timing specifications, the algorithm returns to the beginning of the AFTB partitioning loop (line 8) or, if AFTB partitioning is exhausted, to the beginning of the entire synthesis algorithm for re-tiling (line 5) The next iteration of line 6 partitions the design giving more slack (i.e. free CLBs) to the area of the previous iteration's failing tile If no other tiling

possibilities exist, the algorithm returns to line 1 to generate a new base design If no base design possibilities remain, the algorithm terminates as unsuccessful

The synthesis approach is illustrated using the PREP 5 benchmark shown in Figures 3-5 Figure 3 shows an implementation of the PREP 5 benchmark on the Xilinx 4000 architecture, a configuration that occupies rows 18-24 and columns 1-4 Only the CLBs in the defined area are used in the tiled design; the remaining CLBs are prohibited from use in any of the AFTBs

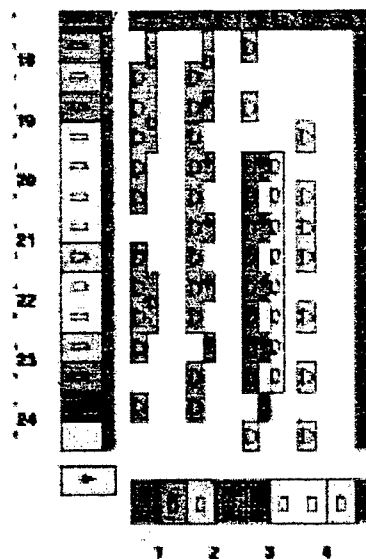


Figure 3: Initial floorplan for PREP 5 benchmark

Tile A covers rows 18-24 and columns 1-2, and tile B covers rows 18-24 and columns 3-4 Tiling was restricted by the occurrence of hard macros in each tile

After partitioning the design into a set of tiles, the tiles are ordered by their implementation difficulty In Figure 3, the tiles are ordered A first and B second, primarily because the logic in tile A is denser

Next we create a set of AFTBs for each tile, which are sorted in decreasing order of implementation difficulty The tiles in Figure 3 are assigned AFTBs that each possess two adjacent free CLBs Each CLB is covered only once, making the total number of AFTBs per tile seven For each AFTB, a complete configuration (i.e. one AFTB for each tile) is passed through the place and route tool Although the placement and routing of the logic in the tile can be quite different for each AFTB, variations within a tile do not propagate to other tiles (other than timing) This is possible because, for each fault detected, only the tile in question is changed All other tiles remain the same as in the original configuration

Figure 4 shows the AFTB that was attempted first for the design in Figure 3

5.3 Enforcing Fault-Tolerance at Run-time

After all of the AFTBs are stored in memory and the circuit begins operation, the system runs normally with the original configuration until a fault is detected. Upon detection, the circuit ceases functional mode until the proper reconfiguration can be made. As already mentioned, we assume that the fault detection system is able to identify the faulty resource in an architecture map. This information allows the system to retrieve the appropriate AFTBs from the configuration memory. The time needed for this memory access depends on normal access factors: access size, memory bus width, memory size, etc.

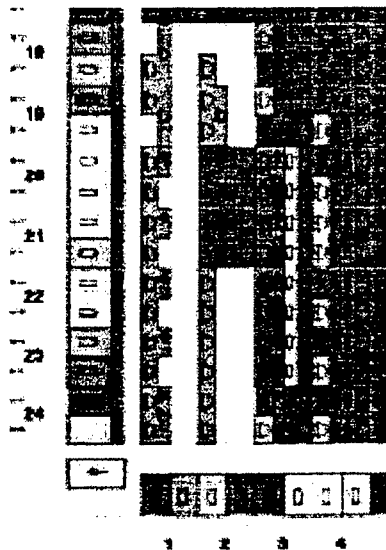


Figure 4: PREP benchmark 5 after tiling with one AFTB

The last step involves the actual reconfiguration of the FPGA device. Once the AFTB is retrieved from memory, two options are possible depending on the capabilities of the FPGA architecture. If the device supports partial reconfiguration, the AFTBs of the affected tiles can be used in isolation to directly update the configuration. Otherwise, the AFTBs must be merged with the active and functioning AFTBs, thus providing the necessary data for a total chip reconfiguration. For example, if the CLB at row 20, column 3 failed in the design of Figure 3, the proper configuration for tile B would be fetched from memory and configured onto the FPGA. The result is shown in Figure 5. Note that the interface between tiles A and B is unchanged.

The time required to update a tile with a new AFTB depends on the complete set of tiles, the device architecture and the surrounding computer system. However, in all cases, it is bounded and can be used to provide a measured level of system availability. After updating the affected tile, the device is reset and the system resumes operation as before the fault. The only possible change could be in the timing of the circuit, as the routing in the altered tiles has likely changed. Timing numbers are generated with each AFTB, and the system can operate under worst case

assumptions. Another, technically more demanding, option is the use of a programmable clock. If new faults occur later, the process repeats itself until more than one fault occurs in a tile for which there is no available AFTB that has all faulty CLBs as unused.

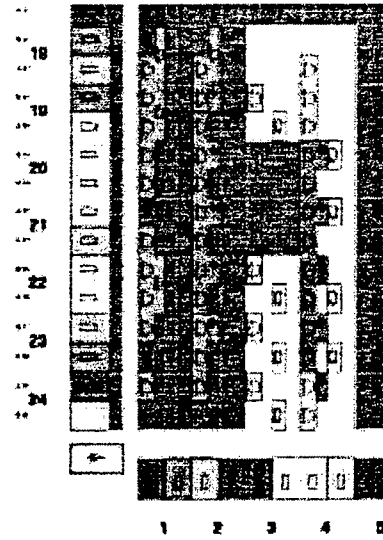


Figure 5: System at runtime after swapping the AFTB in tile B due to fault at (20,3)

6. Reliability Calculation

The first step in calculating reliability is the selection of fault models. There are two major sources of logic faults in FPGA systems: cosmic radiation and manufacturing/operating imperfections.

Since the size of radiation particles is usually small when compared to the size of modern FPGA CLBs, we selected a cosmic radiation fault model that follows uniform distribution of independent (non-correlated) failures. Extensive terrestrial efforts to accurately model the rate of such soft faults indicate high variance (several orders of magnitude) depending on factors such as seasonal solar activity, altitude, latitude, device technology, and device materials. Even for the same chip from the same manufacturer, variations by a factor higher than 200 are not uncommon [25]. Experiments indicate that in FPGA-like devices at an altitude of 20 km, error rates significantly higher than once per 1000 hours are common [15]. Also, as circuit devices become smaller, they become more sensitive to soft faults [25]. If one considers the multiyear life of computing devices and other sources of potential errors (e.g. power surges), the need for fault tolerance in devices which implement critical functions becomes apparent.

The second class of faults is related to manufacturing imperfections. These defects are not large enough to impact initial testing, but after a longer period of operation they become exposed. Design errors can also cause a device to stop functioning in response to rare sequences of inputs (e.g. due to a power density surge in a small part of

the design). For this type of model, we follow the gamma-distribution Stapper fault model [20]. The model is applicable on any integrated circuit with regular repetitive structure, including memories and FPGA devices.

In the remainder of the section, we elaborate on technical details related to the two fault models' reliability calculations.

6.1 Independent Uniformly Distributed Faults

Suppose that a design is partitioned into t tiles. Furthermore, assume that tile i has a total of c_i AFTBs.

The total number of used CLB in the initial design is t_n .

For the sake of clarity and simplicity, we limit our discussion to the case where each AFTB consists of three or fewer CLBs. We denote by m_{1i} , m_{2i} , m_{3i} the number of AFTBs of size one, two, and three CLBs respectively in tile i . We also denote their weighted sum $m_i = m_{1i} + 2 * m_{2i} + 3 * m_{3i}$.

Finally, we assume that the probability of a CLB being faulty is $(1 - P)$ (i.e. the probability that a CLB is fault free is P). It is easy to see that the probability, P_{init} , that the original design is fault free is $P_{init} = P^{m_i}$.

It is also easy to verify that the probability that a tile i in the optimized design is fault free, Pft_i , is given by the following formula:

$$Pft_i = P^{m_i} + m_i * P^{(m_i-1)} (1 - P) + (m_{2i} + 3 * m_{3i}) * P^{(m_i-2)} * (1 - P)^2 + m_{3i} P^{(m_i-3)} * (1 - P)^3$$

The first term corresponds to the scenario where all CLBs are fault free. The last three terms correspond to the scenarios where one, two, and three CLBs are faulty, respectively. The probability (Pft) that the optimized

fault-tolerant design is functional is $Pft = \prod_{i=1}^t Pft_i$.

6.2 Stapper's Fault Model

To calculate reliability for correlated faults, we started from the following formula [20]:

$$\bar{Y}_{mii} = \binom{n}{m} \bar{Y}_1^m \left(\prod_{i=0}^{m-1} \frac{\mu + i}{\mu + i \bar{Y}_1} \right) \times (1 - \bar{Y}_1)^{n-m} \left(\prod_{j=0}^{n-m-1} \frac{\mu + j \bar{Y}_1 / (1 - \bar{Y}_1)}{\mu + m \bar{Y}_1 + j \bar{Y}_1} \right)$$

(Note that we have fixed a small typographical error in the original published formula.)

The formula calculates the probability that exactly m out of n identical modules operate correctly for a given value of the variability parameter μ and single CLB reliability Y_1 . The parameter μ indicates the assumed or the measured probability of clustered faults. Small values of μ imply

high levels of clustering. As μ tends toward infinity the formula reduces to the case of independent uniformly distributed faults.

Stapper's formula is used to calculate the probability that at least m out of n modules (in this case CLBs) operate correctly by a direct summation of relevant terms. This helps reveal how effective this fault-tolerance approach is in the face of several clustered CLB faults.

7. Experimental Results

We conducted an evaluation of the proposed approach and optimization algorithms in two phases. In the first, we applied the approach to nine MCNC designs. In the second phase we studied expected reliability improvement trends as a function of the number of used CLBs in a design.

Tables 1 and 2 show timing and cost (area) metrics respectively of the designs before and after the application of the new approach for reliability enhancement. The first column in both tables indicates the name of a design. The next four columns in Table 1 show the initial delay, and the best, worst, and median delay of the optimized designs. The rightmost column indicates the timing overhead as a result of enhanced reliability. For all nine designs, the largest timing overhead was in the range of 14% to 45%.

A number of factors complicate the task of calculating the physical resource overhead. The place-and-route tools indicate the number of CLBs that are used for a particular placement. However, these utilized CLBs rarely are packed into a minimal area. Unused CLBs introduce flexibility into the place-and-route step that may be essential for completion or good performance. For example, the initial c880 design has a concave region that contains 42 utilized CLBs but also 10 unutilized CLBs (19%). Therefore, we will report overhead in terms of the area used by the fault-tolerant design minus the total area of the original design, including unused CLBs such as the 19% measure above. The area overhead is presented in Table 2, using the same format as Table 1. The average, median and worst-case area overheads were 5.4%, 5.3%, and 9.8% respectively.

Table 3 shows reliability improvements for the MCNC benchmarks under the uniform random fault model. The first column indicates the assumed probability (p) that a CLB is fault free. The next two columns show the probability that the original and fault-tolerant design of a particular benchmark is functioning properly. For example, for 9sym, with $p = 0.995$, the probability of the initial design and tiled design being functional is 81.0% and 98.4% respectively.

Table 4 shows the reliability figures for the same set of designs (original and tiled) with four different variability factors, μ , assuming the probability that a CLB is fault free is 90% and 99%. Table 5 is, in a sense, the strongest indication of the effectiveness of the proposed approach for

Design	Initial (ns)	Fastest (ns)	Slowest (ns)	Median (ns)	<u>Slowest – Fastest</u> Fastest
9sym	71.6	71.6	82.0	76.8	0.15
c499	104.9	104.9	130.0	113.6	0.24
c880	110.8	110.8	126.4	117.3	0.14
duke2	87.9	87.9	118.8	96.4	0.35
rd84	50.2	50.2	72.8	58.6	0.45
planet1	145.0	145.0	194.9	166.1	0.34
styr	150.6	150.6	189.8	167.2	0.26
s9234	135.0	135.0	183.6	153.2	0.36
sand	97.6	97.6	117.7	103.8	0.21

Table 1: Timing bounds due to routing variation among AFTBs for each tile

Design	Original # of CLBs	Final # of CLBs	<u>Final - Original</u> Original
9sym	46	49	.065
c499	94	96	.021
c880	110	115	.045
duke2	93	100	.075
rd84	27	28	.037
planet1	95	100	.053
styr	78	81	.038
s9234	195	206	.056
sand	82	90	.098

Table 2: Variation of resources used among AFTBs for each tile

CLB P	.900		.950		.990		.995		.998		.999		.9999	
Design	Orig.	Tiled	Orig.	Tiled	Orig.	Tiled	Orig.	Tiled	Orig.	Tiled	Orig.	Tiled	Orig.	Tiled
9sym	1.2	16.9	11.6	56.2	65.6	95.7	81.0	98.4	91.9	99.5	95.9	100	99.2	100
c499	0.01	1.63	3.2	37.5	43.0	89.0	65.6	95.6	84.5	98.6	91.0	99.3	99.2	100
c880	0.0	0.6	1.8	31.7	37.3	91.2	61.2	97.6	82.2	99.6	90.7	99.9	99.0	100
duke2	0.01	0.7	2.8	31.9	41.3	90.8	64.3	97.5	83.8	99.6	91.6	99.9	99.1	100
rd84	7.2	38.6	27.7	74.7	77.8	98.5	88.2	99.6	95.1	99.9	97.5	100	99.8	100
planet1	0.02	5.4	11.5	32.6	41.7	94.1	64.7	98.4	84.0	99.7	91.7	99.9	99.1	100
styr	0.01	2.9	2.5	31.4	48.5	93.8	69.7	98.3	86.6	99.7	93.0	99.9	99.2	100
s9234	0.0	0.003	0.01	3.17	16.1	82.0	40.2	94.8	69.5	99.1	83.3	99.8	98.2	100
sand	0.03	1.53	1.83	2.50	45.7	92.4	67.6	97.9	85.5	99.7	92.5	99.9	99.2	100

Table 3: Reliability of the original vs. tiled designs against CLB reliability

	1		2		5		20	
	Orig.	Tiled	Orig.	Tiled	Orig.	Tiled	Orig.	Tiled
9sym	62.6/95.8	72.0/97.2	48.5/93.5	62.8/96.4	28.5/89.0	40.1/94.9	8.77/79.6	29.7/94.1
c499	57.9/95.1	68.2/96.5	41.6/92.2	54.8/95.5	19.9/86.1	36.2/94.1	2.76/71.6	14.5/88.7
c880	55.9/94.9	65.7/96.3	40.2/91.9	53.2/95.2	18.3/85.4	33.8/93.3	2.08/69.8	11.2/87.2
duke2	57.6/95.0	67.9/96.3	41.1/92.1	54.2/95.4	19.4/85.9	25.9/93.9	2.54/71.0	14.3/88.4
rd84	66.4/96.3	76.7/97.7	54.3/94.5	70.1/96.3	36.9/91.1	56.4/95.6	18.0/85.1	52.8/95.1
planet1	57.7/95.0	67.9/96.3	41.3/92.2	54.2/95.4	19.5/86.0	25.9/93.9	2.59/71.2	14.3/88.4
styr	58.9/95.2	68.1/96.9	43.0/92.5	56.2/95.8	21.6/86.8	39.6/94.4	3.63/73.4	15.4/89.8
s9234	53.1/94.3	64.4/95.6	35.1/90.9	58.9/93.5	13.1/82.9	28.4/89.6	0.63/62.5	5.32/83.6
sand	58.4/95.1	68.0/96.6	42.3/92.3	55.3/95.6	20.7/86.4	32.1/94.2	3.15/72.3	14.8/89.2

Table 4: Reliability of original and tiled designs using Stapper's correlated failure model with CLB reliability of 90%/99%

	CLB Overhead for Tiling	Random Fault Model		Stapper's Fault Model	
		Orig.	Tiled	Orig.	Tiled
9sym	6.5%	2.4	16.9	16.8	29.7
c499	2.1%	0.02	1.6	5.4	14.5
c880	4.5%	0.00	0.6	4.1	11.2
duke2	7.5%	0.02	0.7	5.0	14.3
rd84	3.7%	13.9	38.6	32.8	52.8
planet1	5.3%	0.04	5.4	5.1	14.3
styr	3.8%	0.02	2.9	7.1	15.4
s9234	5.6%	0.00	0.003	1.3	5.32
sand	9.8%	0.06	1.5	6.2	14.8

Table 5: Comparison of reliability and overhead for original design with complete redundancy (i.e. 100% overhead) vs. tiled design for CLB reliability of 90% and $\mu = 20$

CLB	100 CLB design		1000 CLB design		5000 CLB design	
Reliability	Orig.	Tiled	Orig.	Tiled	Orig.	Tiled
.9500	0.005921	0.444669	0.000000	0.000302	0.000000	0.000000
.9750	0.079551	0.800119	0.000000	0.107534	0.000000	0.000014
.9800	0.132687	0.864375	0.000000	0.232820	0.000000	0.000684
.9850	0.220739	0.919633	0.000000	0.432660	0.000000	0.015161
.9900	0.366277	0.962643	0.000043	0.683364	0.000000	0.149026
.9950	0.606224	0.990317	0.006704	0.907280	0.000000	0.614762
.9980	0.819220	0.998429	0.136145	0.984404	0.000047	0.924414
.9990	0.905528	0.999608	0.370696	0.996091	0.007000	0.980610
.9995	0.951999	0.999903	0.611453	0.999028	0.085470	0.995153
.9999	0.990868	0.999995	0.912346	0.999952	0.632119	0.999762

Table 6: Reliability of traditional design methods vs. tiled approach against CLB reliability for large FPGAs

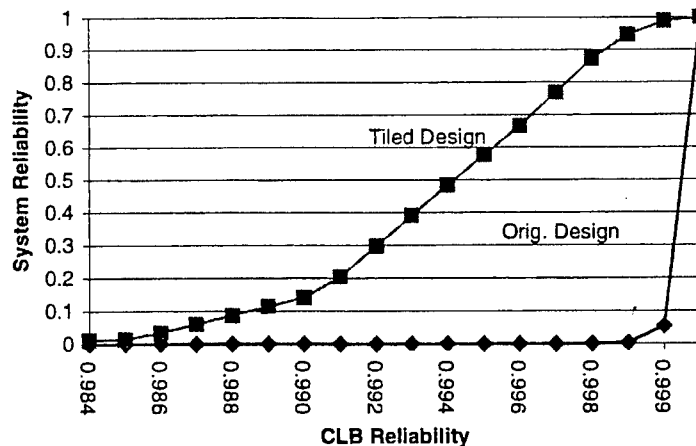


Figure 6: Reliability of traditional methods vs. tiled methods for a hypothetical 5000 CLB FPGA

reliability improvement. The second column of this table indicates the area overhead of tiled designs. The next four columns provide reliability data under two selected fault models for the duplication-enhanced fault-tolerant original and for the tiled designs. For both models, the tiled designs have significantly lower area overhead and always higher reliability than the conventional fault-tolerant designs.

Finally, Table 6 calculates reliability improvement trends as the size of designs increase. It is assumed that all designs are partitioned into tiles of 5 AFTBs each consisting of two CLBs and requires an average hardware overhead (i.e. ~5.4%). Table 6 indicates the potential of the proposed approach for reliability enhancement. For example, in the case of a 5000 CLB design, with $p = 0.999$, the probability of the initial design being functional is less than 1%, while the probability of the tiled design being functional is 98%. Figure 6 graphs the reliability results for the 5000 CLB design.

8. Future work

Many of the highest volume FPGA devices tend to be dominated by interconnect resources, e.g. the Xilinx 4000 and the Altera 10K families. On the Xilinx 4000EX series, the majority of configuration bits are used to program the state of the interconnect rather than the CLBs, and it is likely that these interconnect resources are more susceptible to faults. The fault-tolerance methodology presented above addresses faults in interconnect resources directly dedicated to specific CLBs because they appear as CLB faults. Unfortunately, the vast majority of interconnect resources pass through higher-level hierarchical switch structures that are not covered by unique CLB faults. Some of these routing resources will remain unused in each AFTB, thus providing some additional fault-tolerance. However, since this benefit comes as a byproduct of the approach rather than as a primary goal, we currently cannot make any specific claims on interconnect fault tolerance.

9. Conclusions

Fault-tolerant techniques have recently emerged as an important design consideration for FPGA-based systems due to the rapid progress in FPGA integration and the growing market for these devices. In order to address this problem, we have developed the first fault-tolerance approach to work at the level of physical design. Our hierarchical fault-tolerance technique partitions designs into tiles and atomic fault-tolerant blocks. The approach scales systematically through an exploration of the design solution space at the physical level. The approach is constructed of four phases: design partitioning, tile partitioning and ordering, AFTB partitioning and ordering, and reliability calculation.

Experimental results conducted on a subset of the MCNC benchmarks for large CLB FPGAs indicate that the technique is effective with low hardware overhead.

10. Acknowledgments

The authors would like to thank Prof. Jason Cong, John Peck, Hea Joung Kim, and Jason Leonard for their assistance. This work was supported by the Defense Advanced Research Projects Agency of the United States of America, under contract DAB763-95-C-0102 and subcontract QS5200 from Sanders, a Lockheed Martin company.

11. References

- [1] Abramovici, M., et. al. *Digital Systems Testing and Testable Designs*, New York, Computer Science Press, 1990.
- [2] Carter, W. S., et. al. "A User Programmable Reconfigurable Logic Array", *Proceedings of the Custom Integrated Circuits Conference*, pp. 233-235, 1986.
- [3] Chen, X. T., et. al. "A Row-Based FPGA for Single

- and Multiple Stuck-At Fault Detection," *IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, 1995
- [4] Greene, J W and A E Gamal, "Configuration of VLSI Arrays in the Presence of Defects", *Journal of the ACM*, vol 31, no 4, pp 694-717, 1984
 - [5] Hanchek, F and S Dutt, "Node-Covering Based Defect and Fault-Tolerance methods for Increased Yield in FPGAs", *Proceedings of the Ninth International Conference on VLSI Design*, pp 225-229, 1995
 - [6] Hassan, N and C L Liu, "Fault Covers in Reconfigurable PLA's", *Proceedings of the International Conference on Fault-Tolerant Computing*, pp 166-173, 1990
 - [7] Howard, N J, et al "The Yield Enhancement of Field-Programmable Gate Arrays", *IEEE Transactions on VLSI Systems*, vol 2, pp 115-123, 1994
 - [8] Huang, W K and F Lombardi, "An Approach for Testing Programmable/Configurable Field Programmable Gate Arrays," *IEEE VLSI Test Symposium*, 1996
 - [9] Kikuda, S , "Optimized Redundancy Selection Based on Failure-Related Yield Model for 64-Mb DRAM and Beyond", *IEEE Journal of Solid State Circuits*, vol 26, no 11, pp 1550-1555, 1991
 - [10] Koren, I and D K Pradhan, "Introducing Redundancy into VLSI Designs for Yield and Performance Enhancement", *International Conference on Fault-Tolerant Computing*, pp. 330-335, 1985
 - [11] Levitt, K N, et al "A Study of the Data Communication Problems in Self-Repairable Multiprocessors", *Conference Proceedings of AFIPS*, Washington, D C, Thompson Book, pp 515-527, 1968
 - [12] Mathur, A and C L Liu, "Timing Driven Placement Reconfiguration for Fault-Tolerance and Yield Enhancement in FPGAs", *Proceedings of the ED&TC 96*, pp 165-169, 1996
 - [13] Michinishi, H, et al "A Test Methodology for Configurable Logic Blocks of a Look-up Table Based FPGA," *Transactions of the Institute of Electronics, Information and Communication Engineers*, vol J79D-I, pp 1141-1150, 1996
 - [14] Moore, W R, "A Review of Fault-Tolerant Techniques for the Enhancement of Integrated Circuit Yield", I, pp 684-698, 1986
 - [15] O'Gorman, T J, et al "Field Testing for Cosmic Soft-Error Rate", *IBM Journal of Research and Development*, vol 40, no 1, pp 51-72, 1996
 - [16] Rose, J, et al "Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency", *IEEE Journal of Solid State Circuits*, vol 25, pp 1217-1225, 1990
 - [17] Sarrazin, D B and M Malek, "Fault-Tolerant Semiconductor Memories", *IEEE Computer*, vol 17, no 8, pp 49-56, 1984
 - [18] Shnidman, N, W H Mangione-Smith, and M Potkonjak, "Fault Scanner for Reconfigurable Logic," *Advanced Research in VLSI*, Ann Arbor, MI, 1997
 - [19] Siewiorek, D P and R S Swartz, *Reliable Computer Systems Design and Evaluation*, Burlington, MA, Digital Press, 1992
 - [20] Stapper, C H, "A New Statistical Approach For Fault-Tolerant VLSI Systems", *The Twenty-Second International Symposium on Fault-Tolerant Computing*, pp 356-365, 1992
 - [21] Stroud, C, et al "Built-In Self-Test of Logic Blocks in FPGAs (Finally, a Free Lunch: BIST Without Overhead!)," *IEEE VLSI Test Symposium*, 1996
 - [22] Tanabe, A, et al "A 30-ns 64-Mb DRAM with Built-in-Self-Test and Self-Repair Functions", *IEEE Journal of Solid State Circuits*, vol 27, no 11, pp 1525-1533, 1992
 - [23] Wey, C L, et al "On the Design of a Redundant Programmable Logic Array (RPLA)", *IEEE Journal of Solid-State Circuits*, vol 22, no 1, pp 114-117, 1987
 - [24] Xilinx, *The Programmable Logic Data Book*, San Jose, CA: 1996
 - [25] Ziegler, J F, et al "IBM Experiments in Soft Fails in Computer Electronics (1978-1994)", *IBM Journal of Research and Development*, vol 40, no 1, pp 3-18, 1996

A Mathematical Benefit Analysis of Context Switching Reconfigurable Computing

Stephen M. Scalera
Sanders, A Lockheed Martin Company
PTP2-B007
65 River Road
Hudson, NH 03051
stephen.m.scalera@lmco.com
Phone: (603) 885-0679 FAX: (603) 885-7623

Dr. John J. Murray & Steve Lease
Signal Systems Corporation
294 Tolstoy Lane
PO Box 787
Severna Park, MD 21146-0787
ssc@aol.com
Phone: (410) 431-7148 FAX: (410) 431-8884

Abstract

Dynamic reconfiguration of field programmable gate arrays (FPGAs) has recently emerged as the next step in reconfigurable computing. Sanders, A Lockheed Martin Company, is developing the enabling technology to exploit dynamic reconfiguration. Sanders, working with Signal Systems Corporation, identified key elements to the successful utilization of context switching. Two applications, time-domain beamforming and optical flow, are described in an attempt to reveal some of the inherent computational enhancements afforded by context switching. Conclusions are drawn as a result of this work and future work is described.

1 Introduction

Previous research has examined performance enhancement available from run-time reconfiguration. Current reconfiguration time, milliseconds, although acceptable for some applications, such as the Sanders *SPEAKeasy* reconfigurable "softradio", is unacceptable for many real-time systems. Partial reconfiguration reduces reconfiguration time additionally. Complete FPGA reconfiguration at a rate that far exceeds the necessary persistence of a hardware function is believed to be tomorrow's reconfigurable computing model. Dynamic FPGA reconfiguration, which permits data sharing between configurations, is referred to as *context switching reconfigurable computing* (CSRC).

A context switching FPGA is currently being developed at Sanders. Several computational models are suggested to exploit the features of a CSRC FPGA, however, it is believed that the true potential of CSRC requires a shift in approaches to algorithm implementation. The capabilities of the CSRC architecture have the potential to support improved implementations of signal processing algorithms over current generation mathematical approaches. The emphasis of this paper is to reveal the results of a mathematical benefit analysis of context switching on a CSRC FPGA. In this study, candidate algorithms are assessed to reveal potential benefits afforded by a CSRC implementation. Two of these candidate algorithms, time domain beamforming and optical flow, are analyzed in detail to size the applications to CSRC components and to evaluate the benefits of CSRC technology over current reconfigurable computing (FPGA) devices. Conclusions are advanced in section 4.

2 Time Domain Beamforming

At the core of sonar and radar signal processing is the notion of beamforming. Incoming waves (acoustic or electromagnetic) are received by an array of sensors and the sensed signals are digitized and processed to

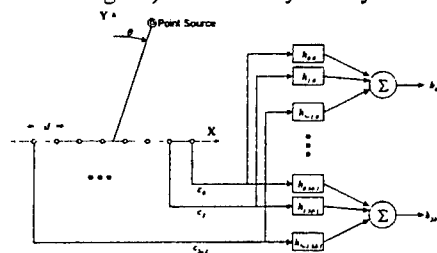


Figure 21 Time Domain Beamforming

determine the direction of the source of the incoming waves. Figure 2.1 shows a line array of uniformly spaced sensors in the presence of a point source. The point source is taken to be distant enough so that circular effects are negligible. Thus, when the wavefront arrives at the line array it is planar, and the angle of arrival θ , is measured with respect to the normal of the line array. Under these conditions, the signals on each channel of the line array are related to one another by simple time delays. $c_i(t) = c_0(t - \tau_i)$, where $c(t)$ is the continuous time

signal sensed on channel i and $\tau_i = i \frac{d}{c} \sin \theta$

As shown in the diagram, d is the distance between sensors, θ is the angle of arrival of the incoming wave, and c is the speed of propagation of the wave. This establishes a fundamental functional relationship between the wave's angle of arrival and the time of arrival of the wave at each channel sensor. Time domain beamforming passes the channel signals through delay filters, which approximate time delays, to temporally align them so that a

coherent summation of the delayed channels amplifies signals associated with waves arriving from a particular angle. A coherent summation of N delayed channels that amplifies the signals from the point source in our example of Figure 2.1 is $\sum_{i=0}^{N-1} c_i(t - \tau_{N-1-i})$

By changing θ , and consequently the delays τ , we can change the direction in which the above summation looks. Moreover, as illustrated in Figure 2.1, the channel data can be processed in parallel to produce M beams, each with a different look angle. In particular, the j^{th} beam which has look angle θ_j is the coherent summation

$b_j(n) = \sum_{i=0}^{N-1} c_i(t - \tau_{N-1-i})$, where the time delays are given by $\tau_{i,j} = i \frac{d}{c} \sin \theta_j$. In Figure 2.1, the time delays are approximated by realizable filters $h_{i,j}$, each approximating a delay of $\tau_{i,j}$. A digital time domain beamforming algorithm using finite impulse response (FIR) filters $h_{i,j}(n)$ is expressed as $b_j(n) = \sum_{i=0}^{N-1} \sum_{l=0}^{L-1} h_{i,j}(l) c_i(n-l)$, where n is the discrete time sampling index, T denotes the sampling period, and L is the length of the FIR filters, which must be large enough to account for the longest time delay needed. $L > \max_i \left(\frac{\tau_{i,j}}{T} \right)$

In practice, the time delay FIR filters will only have K non-zero coefficients which interpolate the appropriate data samples to approximate delays which are not integer multiples of the sampling period. Thus, the evaluation of each FIR filter requires only K multiplies (rather than L). Additionally, the beamforming FIR filters also exhibit channel symmetry: $h_{i,j}(n) = h_{N-1-i,j}(L-n-1)$ which can be exploited to halve the number of multiplies

as long as the data from both channels i and $N-1-i$ are available: $b_j(n) = \sum_{i=0}^{N/2-1} h_{i,j}(l) (c_i(n-l) + c_{N-1-i}(n+L-1-l))$, for even

N . Finally, we note that the above symmetry is usually maintained even when the beamforming FIR filter coefficients are shaded with, for example, Taylor windows.

The time domain beamforming algorithm is thus naturally parallel in that the computations for each beam may be done simultaneously. It can be further segmented by evaluating the beamforming equation over subsets of channel indices to generate partial beams, which are then summed to complete the beams. The partial beam

for channel i and beam j is: $p_{i,j}(n) = \sum_{l=0}^{L-1} h_{i,j}(l) c_i(n-l)$. Subbeams are formed when partial beams are summed over a subset I of the available channels.

$s_{I,j}(n) = \sum_{i \in I} p_{i,j}(n)$ The fact that the beamforming expression lends itself naturally to parallel implementations allows for a simple mapping onto the CSRC hardware which exploits symmetry and identically zero FIR filter coefficients for computational advantage.

Each CSRC FPGA stores N_f channel time histories of length L in configurable logic blocks (LCs) which are configured as RAM accessible from all context layers. Thus, F CSRC FPGAs can store the time histories for $N = N_f F$ channels. Based on the number of remaining LCs, each of the C context layers can process M_f subbeams. Thus, a single CSRC FPGA can compute $M = M_f C$ subbeams associated with its N_f channel inputs. Figure 22 illustrates this decomposition for a single CSRC FPGA, where the input channel vector $C_{i(f)}(n)$ is the set of N_f channel histories $c_i(n)$ for which $i \in I(f)$. The FPGA index i_f thus determines which group of channel histories is presented to each of the CSRC FPGAs, $i_f = 0, 1, \dots, F-1$. In order to exploit the channel symmetry discussed earlier for computational advantage, the number of channel histories stored on each CSRC FPGA, N_f , must be even. Furthermore, channels i and $N-1-i$ must both be contained in one of the groups $I(i_f)$.

A system consisting of F CSRC FPGAs, each receiving N_f channels, can thus process $N = N_f F$ channels into $F M_f C$ subbeams, which are then summed to produce $M_f C$ beams. If a greater number of beams is required, this architecture can simply be replicated (any number of times), with each such system capable of producing $M_f C$ independent beams. Operationally, each CSRC FPGA is presented with the current sample for each of its input channels simultaneously. These inputs are pushed into the time history buffers (tap delay lines) in the first context layer, which also computes the first set of subbeams. The FPGA then sequences through the

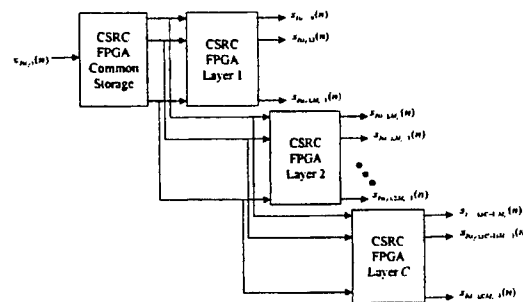


Figure 22: Subbeam decomposition onto CSRC FPGA

remaining context layers to produce the rest of the subbeams. Each context layer sends its output to the same set of I/O pins, so that the subbeams are multiplexed on these lines.

A slight variation to the architecture described above is to push new samples into the channel history buffers in all context layers (not just the first context layer). Each beam would thus be computed every C samples, (but the samples are now arriving C times faster). The advantage of this architecture is that the control lines which specify the active context layer, and hence the output beams, can be commanded in any order. This provides a

hardware device that fits nicely into typical sonar and radar sensing operations which command a search pattern until a target is detected, and then track the target by selecting beams immediately around it.

Table 2.1 performs sizing calculations for a single context layer of a CSRC FPGA for the beamforming application. Using two input channels and two subbeams per context layer keeps the implementation below 80 percent utilization since each context layer has 1024 logic cells (LCs) in the current Sanders design. The calculations in the table include taking advantage of the beamformer

Number of Channels	2	2
Number of Subbeams per Context layer	3	2
Length of Channel Histories	256	256
Length of Beamforming FIRs	4	4
Channel and FIR Coefficient Data Width (bits)	8	8
Subbeam Data Width (bits)	16	16
CLBs per RAM bit	0.0625	0.0625
CLBs per Channel Data Width Multiply	40	40
CLBs per One Bit Addition	1	1
CLBs allocated for RAM	256	256
CLBs allocated for Multiplies	480	320
CLBs allocated for Additions	288	192
Total CLBs Required	1024	768

Table 2.1: CSRC Context Layer Sizing For Beamforming

channel symmetry and the savings that accrue from configuring multiplies where one operand is known a priori and is hard wired in the context layer

3 Optical Flow

The goal of the plume detection DARPA challenge problem is to segment a plume from the background in gray scale video frames. The proposed approach is to assume that the plume is the only moving object in the frames. Using an optical flow algorithm to estimate velocities within the frame, the plume can be segmented by thresholding the estimated velocities. The plume detection problem seeks architectures which can process 256 by 256 images with 12-bit pixels at a frame rate of approximately 1 kHz.

3.1 Optical Flow Algorithm Description

Using the assumption of constant illumination, the optical flow algorithm considers any changes in pixel intensity in time to be caused by object motion. The intensity change due to object motion can be expressed as, $I_t = -u(x, y, t)I_x - v(x, y, t)I_y$, where I is the image intensity, u and v are the x and y velocities of objects within the image, $I_t = \partial I(x, y, t) / \partial t$, $I_x = \partial I(x, y, t) / \partial x$, and $I_y = \partial I(x, y, t) / \partial y$. Equation 3-1 does not account for second order effects at moving object boundaries. Employing estimates \hat{u} and \hat{v} for the unknown object velocities, a measure of the accuracy of the estimates is $e(x, y, t) = I_t + \hat{u}(x, y, t-1)I_x + \hat{v}(x, y, t-1)I_y$, where e is the residual of the estimated change in intensity with respect to time. If the image is divided into subimages, the total

error can be expressed as $E = \sum e(x, y, t)$. The subimage error, E , can then be used to update the velocity estimates using the relations $\hat{u}(x, y, t) = \hat{u}(x, y, t-1) - bEI_x$ and $\hat{v}(x, y, t) = \hat{v}(x, y, t-1) - bEI_y$, where b is a learning parameter. The choice of b and subimage size affect the performance and stability of the optical flow algorithm.

3.2 FPGA Architecture

The ability of the CSRC FPGAs to retain memory during context switching makes them well suited for computing the partial derivatives of the optical flow algorithm. By loading data from successive frames, a single FPGA can compute I_x , I_y , and I_t using one layer for each partial derivative. The partial derivative data can then be passed to a second FPGA to perform the error calculation and velocity estimate updates. Figure 3.1 shows a block diagram of a processing block containing two CSRC FPGAs with a shared 128 Kbyte memory. Both FPGAs have access to the RAM and the two FPGAs have data and control connections between themselves. The 64-bit data connections allow four words to be processed in parallel. A mechanism for arbitration of the RAM bus between the FPGAs and external access would be required. Use of dual-port RAM could simplify the arbitration and improve performance.

The sizing of the plume detection problem discussed in later sections indicates that eight of the processing blocks shown in Figure

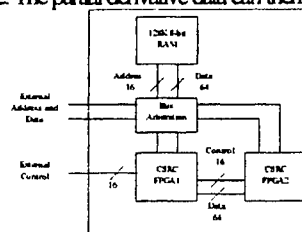


Figure 3.1 CSRC Plume Detection Processing Block

3.1 would be necessary for 256 by 256 pixel images at a frame rate of 1 kHz. Figure 3.2 shows a CSRC architecture block diagram containing eight processing blocks, each with external data and control connections. No communication between the processing blocks would be necessary. The following sections discuss the mapping of the plume detection problem to the architecture shown in Figure 3.2 and the corresponding resource allocation.

3.3 Algorithm Mapping

The plume detection problem can be implemented on the CSRC architecture shown in Figure 3.2 by having the processing blocks operating in parallel on different subimages. Dividing the 256 by 256 image into sixteen 64 by 64 pixel subimages allows each processing block to serially operate upon two subimages. The subimage data is loaded into the processing blocks' RAM from the external interface. The external interface steps through the processing blocks reading the updated velocity estimates and writing the next video frame for the appropriate subimage. Given proper RAM bus arbitration, the subimages can be double buffered to allow the next frame to be loaded while the FPGAs operate on previous frames. A similar double buffering approach could be used for the output velocity estimates.

Table 3.1 describes the mapping of the plume detection problem for a single subimage to the CSRC processing block shown in Figure 3.1. Nine processing steps are listed in Table 3.1 in the order in which they

Step	FPGA1 Layer	Operation	FPGA2 Layer	Operation
1	1	Read partial subimage (16x16) $I(x,y)$ and previous frame partial subimage $I(x,y-1)$ from RAM	1	Compute I_x and keep a running sum of E
2	2	Compute I_x and output I_x to FPGA2 (4 pixels at a time)	1	Compute I_y and keep a running sum of E
3	3	Compute I_y and output I_y to FPGA2 (4 pixels at a time)	2	Add I_x to E and compute bE product
4	4	Compute I_x and output I_x to FPGA2 (4 pixels at a time)	2	
5	Repeat steps 1 through 4 to complete subimage			
6	1	Read partial subimage (16x16) $I(x,y)$ from RAM	3	Update
7	2	Compute I_x and output I_x to FPGA2 (4 pixels at a time)	3	Update
8	3	Compute I_y and output I_y to FPGA2 (4 pixels at a time)	3	Update
9	Repeat steps 6 through 8 to complete subimage			

Table 3.1. CSRC Optical Flow Algorithm Mapping

switches to context layer 3 to compute I_y and FPGA2 continues to use layer 1 to add $\hat{v} I_y$ to E . FPGA1 then switches to layer 4 to compute I_x for step 4 and FPGA2 switches to layer 2 to add I_x to E . Steps 2 through 4 are repeated for each partial subimage until the entire subimage has been processed and the E summation is complete. When the E summation is complete, FPGA layer 2 computes the product bE used for velocity estimate updates. For step 6, FPGA1 reloads context layer 1 and reloads the most recent frame of the partial subimage. FPGA1 then computes I_x and I_y in steps 7 and 8 the same as in steps 2 and 3 and FPGA2 switches to layer 3 to use I_x and I_y to update the velocity estimates which are stored in RAM. FPGA2 could use its fourth layer following step 8 for additional functions such as velocity thresholding or grayscaling. Steps 6 through 8 are repeated until the velocity estimates have been updated for the entire subimage. The FPGA and memory resource utilization for this mapping of the optical flow algorithm are discussed in the following sections.

3.4 Processing Requirements

Table 3.2 lists the number of FPGA clock cycles necessary for each step of the optical flow

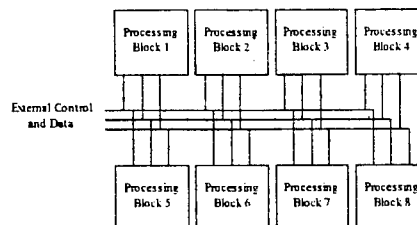


Figure 3.2. CSRC Plume Detection

would be performed and the corresponding operations performed by each FPGA are described. Each step operates upon four words at a time. The two FPGAs have been designated FPGA1 and FPGA2. In step 1, FPGA1 reads two frames of a 16 by 16 partial subimage into the FPGA local memory using its first context layer. For the most recent frame some extra pixels will need to be read to compute I_x and I_y . FPGA1 then switches to context layer 2 for step 2, computes I_x and sends I_x to FPGA2. FPGA2, using its first context layer, receives I_x , multiplies I_x by \hat{u} and begins the summation for E . For step 3 FPGA1

Step	Cycles	Notes (all steps operate on 4 words at a time, memory accesses are assumed to take 2 cycles)
1	$C_1=298$	$((17 \times 3 + 16 \times 4) \times 2)$ Most recent frame requires 17×17 to compute I_x and I_y
2	$C_2=136$	$((16 \times 4) \times 2 \times 8)$ Pipelined operation with 8 cycles of setup and 2 cycles per 4-word vector (2 multipliers)
3	$C_3=136$	$((16 \times 4) \times 2 \times 8)$ Pipelined operation with 8 cycles of setup and 2 cycles per 4-word vector (2 multipliers)
4	$C_4=68$	$((16 \times 4) \times 1 \times 8)$ Pipelined operation with 4 cycles of setup, 1 cycle per 4-word vector and 4 cycles for bE product
5	$C_5=10208$	$16 \times (C_1 + C_2 + C_3 + C_4)$ 64x64 subimage made up of 16 partial subimages
6	$C_6=170$	$((17 \times 5) \times 2)$ Only most recent frame required
7	$C_7=272$	$((16 \times 4) \times 4 \times 16)$ Pipelined operation with 16 cycles of setup and 4 cycles (1 read and 1 write) per 4-word vector
8	$C_8=272$	$((16 \times 4) \times 4 \times 16)$ Pipelined operation with 16 cycles of setup and 4 cycles (1 read and 1 write) per 4-word vector
9	$C_9=11424$	$16 \times (C_6 + C_7 + C_8)$ 64x64 subimage made up of 16 partial subimages

Table 3.2. CSRC Optical Flow Processing Requirements

algorithm. Each 64-bit RAM access is assumed to take two clock cycles. The number of clock cycles required for the pipeline operation consists of two factors. (1) A one time cost for each algorithm step which accounts for the entire length of the pipeline (2) The cost for the slowest part of the pipeline for each 4-word operation. For step 1, FPGA1 reads the most recent subimage frame and the previous frame into memory. Taking into account the 2-cycle, 64-bit reads and the extra data required for the I_x and I_y calculations, step 1 requires 298 cycles for each subimage. In step 2 the slowest part of the pipeline is the calculation of $\hat{u} I_x$. The FPGA only has enough resources for two multipliers, therefore the 4-word multiplication must be done in two steps of two multiplications. With a total pipeline length of 8 cycles and a per vector cost of 2 cycles, step 2 requires 136 cycles for each partial subimage. Functionally, step 3 is identical to step 2. Step 4 does not require multiplies and therefore is assumed to consume 1 cycle per vector with an overhead of 4 cycles and an extra 4 cycles for computing the bE product. The cycle cost of step 5 is the partial subimage cost of steps 1 through 4 times the 16 partial subimages which compose a subimage. Step 6 is similar to step 1 except that the previous frame is not necessary since I_x does not need to be computed. The memory accesses for reading and writing the velocity estimates are the slowest parts of the pipeline for steps 7 and 8. With two memory accesses at 2 cycles per access the per vector cost is 4 cycles. Step 8 computes the cost of steps 6 through 8 for an entire subimage. The total image cost is then computed based upon the subimage costs for steps 5 and 9. Given an 80 MHz FPGA and a 1000 Hz frame rate, 80000 cycles are available per processing block for each frame. Eight processing blocks would provide 640000 cycles per frame which would require each processing block to operate at 54 percent of its estimated capacity. The low estimated utilization allows for errors in estimates, slower FPGA parts or added functionality such as velocity thresholding.

3.5 FPGA Resource Requirements

Table 3.3 lists the number of LCs used in each layer for memory and for arithmetic functions. When a LC is used as memory, the LC is assumed to provide 16 bits of storage.

FPGA	Layer	Required LCs	Computational (1 LC per bit with 300 LC per 12 bit by 12 bit general coefficient multiply)
1	1	543	17*17*16*16
1	2	593	17*17*16*16
1	3	593	17*17*16*16
1	4	593	17*17*16*16
2	1	698	2
2	2	298	2
2	3	613	1
2	4	N/A	

Table 3.3. CSRC Optical Flow Algorithm FPGA Resource

multipliers for the $\hat{u} I_x$ or $\hat{v} I_y$ calculation, four 24-bit adders for the E summation and 24 bits of storage for E .

FPGA2 layer 2 has the same requirements as layer 1 without the multipliers. Without the need for the same multipliers as layer 1, layer 2 is a good place to perform the bE multiplication at the end of a subimage. The bE multiplication can be done as a constant coefficient. Given b is a 12-bit constant and E is 24 bits, the bE multiplication would consume 200 LCs. The third layer of FPGA2 requires two 12-bit multipliers and two 16-bit adders for the velocity estimate updates and 12-bits of storage for bE . The maximum number of LCs used by any FPGA layer is 698.

3.6 Memory Requirements

Table 3.4 lists the RAM requirements for the CSRC optical flow architecture. The optical flow algorithm stores the frame data, I , and the velocity estimate data, \hat{u} and \hat{v} , in the RAM. Additional memory is assumed to be used for double buffering the input frame data and the output velocity estimates. The total memory requirement for each frame is 896 KB. Each of the 8 processing blocks having 128 KB providing a total of 1 MB.

3.7 CSRC FPGA Alternative Architectures

Alternatives to the CSRC FPGA optical flow architecture include ASIC, integer DSP and standard reconfigurable FPGA architectures. Table 3.5 lists the possible optical flow architectures and their respective advantages and disadvantages. The CSRC FPGA architecture implements the 800 Mops optical flow algorithm in real-time while maintaining flexibility for changes to the algorithm. A disadvantage of the CSRC FPGA architecture is that programming of the algorithm would be more difficult than for a general purpose processor such as an integer DSP. An ASIC implementation of the optical algorithm could provide the fastest processing

$I(x,y)$	03072	2*256*256
$E(x,y)$	03072	2*256*256
$\hat{u}(x,y)$	03072	2*256*256
$\hat{v}(x,y)$	03072	2*256*256
$\hat{u}(x,y)$	03072	2*256*256
$\hat{v}(x,y)$	03072	2*256*256
$\hat{u}(x,y)$	03072	2*256*256
$\hat{v}(x,y)$	03072	2*256*256

Table 3.4. CSRC Optical Flow Algorithm RAM Requirements

Architecture	Advantages	Disadvantages
CSRC FPGA	Fast Flexible	Moderately difficult programming
ASIC	Very fast Power and volume efficient	Very difficult programming Expensive hardware Not flexible
Integer DSP	Very flexible Easy programming	Slow
standard reconfigurable FPGA (1 FPGA for each CSRC FPGA layer)	Fast Flexible	Moderately difficult programming Expensive hardware Volume costs
standard reconfigurable FPGA (1 FPGA for each CSRC FPGA)	Flexible	Slow Difficult implementation

Table 3.5. Optical Flow Algorithm Architecture

speed along with being power and volume efficient, however an ASIC architecture would have a large up front cost for design and would not be flexible in regards to changes in the algorithm. Using an integer DSP for the optical flow algorithm would allow for the easiest and most flexible programming, however an integer DSP would only provide about 20 Mops. The integer DSP architecture could achieve the required 800 Mops but would need 40 DSPs, which would be costly.

Two possible approaches exist for using a standard reconfigurable FPGA architecture for the optical flow algorithm. One approach is to utilize one standard FPGA for each CSRC FPGA layer. Using this approach the speed and flexibility of the CSRC FPGA architecture can be achieved, but the hardware and associated cost would be increased by a factor of over three. Alternatively, one standard FPGA can be used for each CSRC FPGA. Since reprogramming a standard FPGA takes approximately 30 ms, which corresponds to 30 frames at a 1 kHz frame rate, reprogramming could not be used in the 1-to-1 FPGA replacement architecture. Without the ability to reprogram, a different mapping of the optical flow algorithm to the FPGAs where the different functions are distributed across FPGAs would be needed. Spreading the functions across FPGAs complicates the data flow and likely creates memory access bottlenecks. Hence, the CSRC FPGA architecture provides a combination of speed and flexibility for the optical flow algorithm which cannot be matched by other architectures.

4 Conclusions

A number of signal processing algorithms have been assessed with regard to their potential to exploit the CSRC hardware architecture for computational advantage. This research indicates that the algorithms with the most potential to benefit from the CSRC architecture are those that share a significant number of memory locations between context layers. This mode of computation embodies the notion of moving the algorithm through the data. The bottleneck in an FPGA-based system is oftentimes the off-chip memory access. In other words, the most expensive part of the algorithm implementation is transferring data on and off chip. This is further burdened by the age-old pin limitation problem. The CSRC architecture allows for reduced cycles of data access by allowing the data to be loaded in *once*, process the data with a context, reconfigure (but retain the data) and continue processing with the second context, the third context, and so forth. Since multiple contexts and the concept of virtual hardware alleviate the requirement of confining the logic to a physical resource size, the data need only be transferred on chip at the start of computation and off chip at the end of computation.

Although history has shown that FPGAs are typically developed to be general in nature, this work suggests that a move towards specific algorithm development will be beneficial with the advent of context switching. Being able to develop fixed coefficient multipliers, key-specific decryption algorithms, or sub-branches of a binary tree classifier algorithm are just a few of the possibilities afforded by CSRC. In essence, if a circuit can be designed that is specific, therefore smaller, typically faster, and utilizing less power, doing so makes sense given that designs can be cached and or swapped in and out in real-time.

5 Future Work

As previously mentioned, Sanders is currently developing a CSRC FPGA that is capable of single cycle reconfiguration as well as data sharing between configuration layers. Future publications will document that effort.

6 Acknowledgments

This effort was supported by DARPA/MTO under contract number F30602-96-C-0350.

7 References

- [1] A. DeHon, "Reconfigurable Architectures for General-Purpose Computing", PhD Dissertation—MIT, 1996
- [2] J. Burns, A. Donlin, J. Hogg, S. Singh, M. de Wit, "A Dynamic Reconfiguration Run-Time System", *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997
- [3] S. Kelem, "Mapping a Real-Time Video Algorithm to a Context-Switched FPGA", Poster Session, *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997
- [4] S. Trimberger, "A Time-Multiplexed FPGA", *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997

Signature Hiding Techniques for FPGA Intellectual Property Protection

John Lach¹, William H. Mangione-Smith¹, Miodrag Potkonjak²

Departments of Electrical Engineering¹ and Computer Science²
The University of California, Los Angeles

Abstract – This work presents the first known attempt to leverage the unique characteristics of FPGAs to protect commercial investments in intellectual property. A watermark is applied to the physical layout of a digital circuit when it is mapped into an FPGA. This watermark uniquely identifies the circuit origin and yet is difficult to detect. While this approach imposes additional constraints, experiments involving a number of large complex designs indicate that the performance impact is small.

1 Introduction

We have developed and evaluated a method for applying cryptographically encoded watermarks to digital designs. The approach is shown to successfully encode long messages on existing designs of moderate to large complexity with little or no impact on circuit performance or resource requirements. By using these messages to encode authorship signatures, we can provide compelling evidence to establish design ownership.

1.1 Motivation

It is generally agreed that the most significant problem facing digital IC designers today is system complexity. Complex systems tend to be assembled using smaller components in order to reduce complexity as well as to take advantage of localized data and control flows. This trend toward partitioning enables design reuse, which is essential to reducing development cost and risk while also shortening design time. Design reuse has been employed by systems designers for years; what is new is that the boundaries for component partitions have moved inside of the IC packages. These reusable modules are commonly referred to as Intellectual Property (IP), as they represent the commercial investment of the originating company but do not have a natural physical representation.

Direct theft is a major concern of IP vendors. It is possible for customers, or a third party, to simply sell an IP block as their own without even reverse engineering the design. Because IP blocks are designed to be modular and integrated with other system components, the thief can simply repackage them without bothering to understand either the architecture or implementation.

This paper presents a novel solution to the risk of direct misappropriation. The essential idea involves embedding a digital watermark, which uniquely identifies the creator, in an IP block. This watermark allows the IP owner to verify the physical layout as their property, in a way that is likely to be much more

compelling than the existing option of verifying the design against a registered database.

1.2 Motivational Example

While the concepts developed here can be applied to a wide range of FPGA architectures, all of the discussion and experimental work will be conducted in the context of the Xilinx XC4000 architecture [13]. These devices are composed of an array of configurable logic blocks (CLBs), each of which contains two flip-flops and two 16x1 lookup tables (LUTs). A hierarchical and segmented routing network is used to connect CLBs in order to form a specific circuit configuration.

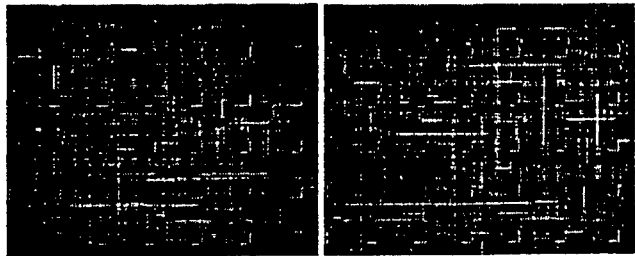


Figure 1a (left). Original design layout

Figure 1b (right). Watermarked design layout

Consider the case of PREP Benchmark #4, a large state machine, which can be mapped into a block of 27 CLBs. This mapping results in 3 unused CLBs, or $3 \times 32 = 96$ unused LUT bits. Each unused LUT bit is used to encode one bit of the signature. Figure 1a shows the layout of the original design as produced by the standard Xilinx backend tools, while Figure 1b shows the layout for the same design after applying the watermark constraints to the three unused CLBs and re-placing the design. The constrained CLBs are then incorporated into the design with unused interconnect and neighboring CLB inputs, further hiding the signature.

1.3 Technological Issues

The standard digital design flow generally follows these steps: behavioral HDL, synthesis to RTL, technology mapping, and finally physical layout involving place and route. Watermarking can be applied to any level of this design flow and, if developed properly, will propagate to later stages [2, 6]. However, because a watermark is fundamentally an optional component of a system design, any watermark can be removed by reverse engineering a design to a stage in the flow before the watermark has been applied. For example, the detailed approach developed here will be used to watermark a design at the physical level by manipulating LUTs and interconnect. The IP vendor will then deliver their technology in the form of a hard macro. If the macro can be reverse engineered to a netlist, the watermark will be removed, specifically because it is not a functional part of the circuit operation. Fortunately, most FPGA vendors have taken a business position that they will not reveal the specification of their configuration streams, specifically to complicate the task of

reverse engineering and thus protect the investment of their customers [10].

1.4 Contributions

This paper presents the first method for protecting Intellectual Property, in the form of reusable digital circuits, even after the IP has been delivered in commercial products. By manipulating hardware resources, we are able to encode relatively long messages in a manner that is difficult to observe by a third party, resists tampering, and has little impact on circuit performance or size. This capability provides three main benefits:

1. It reduces the risk that a watermarked circuit will be stolen, i.e. used illegally without payment or transferred to a third party.
2. It reduces the risk that any unmarked circuit will be stolen.
3. It can be used to identify the backend tool chain used to develop a design, and thus be part of the royalty mechanism used for CAD tools.

2 Reverse Engineering Techniques

While Xilinx and other FPGA vendors make some efforts to complicate the task of reverse engineering, it certainly is possible to recover the configuration specification with a concerted effort. NeoCAD was able to accomplish this for the Xilinx XC4000 series devices through a directed investigation of the bitstreams produced by the Xilinx backend tools. Given this information, it should be relatively straightforward to produce a Xilinx netlist file and then use commercial tools to move back up the design flow. Another possible line of attack involves removing the packaging material and then using a visual inspection tool to produce a circuit representation of the CLB. A similar approach has recently been used to produce a complete layout of a 386 microprocessor in approximately 2 weeks [1].

In response to the proven success of the reverse engineering attacks, we believe that hiding the watermark is necessary but not sufficient. Any effective watermarking scheme should make the signature appear to be part of the functional digital circuit to whatever extent is possible.

3 Related Work

Ad-hoc techniques for the watermarking of text and image documents have been manually practiced for many centuries. Modern techniques for signature data hiding in image, video, and audio signals have received a great deal of attention. A spectrum of steganography-based approaches for protection of digital images has been proposed [3, 9, 12].

Recently, a set of techniques for intellectual property protection through watermarking at the behavioral level down to the physical layout using superimposition of additional constraints in conjunction to those specified by the user has been proposed [2, 5, 6]. In this paper, we propose the first intellectual property protection technique for FPGA designs. Different design phases (physical synthesis of FPGA-based design vs. behavioral synthesis) result in very different sets of synthesis and optimization issues.

Cryptography also has a long history. Two decades ago, the public-key techniques introduced by Stanford researchers redefined the field [4]. Many techniques, from both a practical and theoretical viewpoint, have been summarized in [8].

We use cryptographic techniques to select a subset of FPGA physical design constraints from a set of constraints that are not already used for design specification. An additional benefit is that the cryptographic techniques also provide probabilistic randomization and therefore protection from added constraints. For this task, we use the standard cryptography tools from the PGP-cryptography suite, the secure hash function MD5, and the RSA/MIT stream cipher RC4 [8].

4 Approach

The global flow of our watermarking system is represented by the pseudo-code in Figure 2. First, the complete design passes through the vendor place and route tool in order to get an initial estimate of the resource constraints. The process terminates if the available resources are not sufficient to satisfy the watermark request. In this case, the IP developer has the option of either mapping into a larger physical area or requesting a smaller signature. Next, the signature is transformed in order to make it more difficult to detect and tamper with. Once the signature has been prepared, it is embedded into the input files of the place and route tools, through a combination of netlist modifications and physical constraints. Finally, the modified circuit again is passed through the vendor place and route tools. If the resulting physical layout achieves the system performance goals, then the watermarking process is complete.

1. Read in netlist and desired signature
2. Use vendor tools to place and route unmodified netlist
3. If (not enough spare resources for signature) then exit and retry with smaller signature
4. Process signature:
5. Pack 8-bit ASCII into continuous 7-bit characters
6. Encrypt signature to match "channel", i.e. typical design, spectrum
7. Add error correction coding
8. Interleave ECC-encrypted blocks to combat localized tampering
9. Embed properly-sized clique
10. Modify netlist and physical constraints to embed prepared signature
11. Execute vendor place and route tools on modified netlist
12. If (performance is too low) retry with smaller signature else terminate with success

Figure 2. Global flow of watermarking system

4.1 Signature Embedding

The first step in signature preparation involves transforming the signature so that it will appear to have the same statistics as an actual design. This process can be thought of as an application of encryption, which generally whitens a signal to match a channel with Gaussian white noise. However, in this case, the purpose of whitening the signal is not fundamentally to mask its content but rather its existence.

The next step in signature preparation involves adding error-correction coding (ECC). By doing so, we combat the malicious third party that manages to identify a part of a signature and attempts to modify or remove it. If the modification is small enough and localized, the ECC codes will be useful for retrieving the original signature and providing proof of design tampering.

The final step in signature preparation involves interleaving multiple ECC blocks. It is possible that a malicious third party would be able to identify a particular LUT that is non-essential to

the device function, and change its programming. If sixteen consecutive ECC blocks are interleaved, one bit at a time, over a set of LUTs, then each LUT will only contain one bit from any ECC block. This interleaving guarantees that the validation software can successfully retrieve the signature in the face of any single point fault, i.e. a LUT that has been tampered with.

Embedding the processed signature involves using free LUTs in an unmarked design. Each LUT in the XC4000 family encodes 16 bits of information, and from our experience most designs have a large number of unused LUTs. The signature is coded into LUTs defined by the designer's signature and a secure hash function, and the design is placed and routed around the signature. Since the actual signature is known only to the designers, they are also the only ones who know the location of the unused LUTs. Therefore, if the unused LUT location is disclosed for one design, designs with other signatures are still secure.

4.2 Validation

When the owners of an IP block believe their property has been misappropriated, they must deliver the configuration in question to an unbiased validation team. The IP vendor produces a seed that they claim was used to produce the block. With the seed and signature, the validation team reverses the signature preparation and embedding process: identify the CLB locations used for hiding the signature using the functions defined by the secure hash function, reverse the block interleaving, apply the ECC if necessary, decrypt the message using a known key, and finally print out the resulting signature. If the signature matches that claimed by the IP vendor, then ownership has been established.

5 Experimental Results

We have evaluated the proposed approach by watermarking three large designs on FPGAs with various signature sizes, from an extremely small mark to the maximum size given unused LUT availability.

The overhead of the proposed approach comes in the form of area (physical resources) and timing. Area overhead is inevitable, as previously unused LUTs are used to encode the signature. However, in reality, area overhead does not increase linearly with the size of the signature. Rather, the calculation of area overhead involves the realization that place and route tools rarely pack utilized CLBs into a minimal area. Therefore, area overhead should be viewed in terms of the area used by the watermarked design minus the total area of the original design, including unused CLBs and LUTs.

Timing overhead may arise due to the constraints on placement as defined by the size and location of the signature. A LUT dedicated to the signature may impede placement of circuit components and lengthen the critical path. As the signature size grows, more constraints are made on the placement of the design, thus increasing the possibility for performance degradation.

The three designs used to evaluate the approach are a MIPS R2000 processor core designed for FPGAs, a reconfigurable Automatic Target Recognition (ATR) system [11], and a digital encryption standard (DES) design [7]. The MIPS core and the DES design were both implemented on the Xilinx XC4028EX-3-PG299, and the ATR system was implemented on the XC4062XL-3-PG475. For each design, the smallest possible device was used.

5.1 Results

Experimental results reveal that both area and timing overhead are low. After each design was placed and routed with no signature constraints, the number of unused LUTs was calculated and the circuit timing was noted. The original physical layout statistics are shown in Table 1. In each case, the designs were laid out such that the entire FPGA area was being used, with LUTs and entire CLBs being sporadically unused, illustrating that the place and route tools do not pack logic with optimum density. Therefore, there is essentially no area overhead required by the proposed approach. The approach utilizes free space in the original design and increases the density of occupied CLBs and LUTs. For tools that attempt to pack logic with increased density, area overhead may become apparent depending on signature size.

For each design, incrementally larger signatures were placed in the FPGA, and the design was placed and routed around the restricted resources. For each instance, the circuit timing was noted and compared to the original design. This process was repeated until the largest possible signature, i.e. one making use of all unused LUTs, was implemented. The results are shown in Tables 2-4.

For each table, the top two rows show the size of the watermark, first in bits and then in number of encoded ASCII characters. The next row for each design shows the percent resource increase in terms of the number of used CLBs. As mentioned above, the area increase for each instance is nearly 0%, but the table reflects the additional percentage of CLBs actually utilized in the watermarked design. Finally, the timing degradation for each instance is shown. Positive percentages indicate a decrease in performance. The table reveals that timing degradation is small and even negative in many instances. Relatively small changes in a circuit netlist or routing constraints can often result in a dramatically different placement and a corresponding change in speed. It appears that the impact of watermarking on performance is well below this characteristic variance, and thus the performance impact is non-monotonic with signature size.

Figures 3a and 3b are examples of DES layouts. Figure 3a is the original layout of the design with no watermark constraints. Note that the original placement does not achieve optimal logic density. Instead, unused CLBs are dispersed throughout the design. Figure 3b shows the layout with an embedded signature of 4768 bits.

6 Conclusion

As the market for reusable digital designs grows, issues concerning protection of proprietary designs come to the forefront. This paper has described a technique that takes advantage of FPGA flexibility to encode a watermark that is extremely difficult to detect and/or remove. The watermark uniquely identifies the design's origin, thus protecting designers against misuse or unauthorized distribution. Although the watermark is applied to the physical layout of the design by imposing constraints on the backend CAD tools, the area and timing overhead is extremely low. Experiments have shown that, even on very complex designs, a watermark can be applied and validated at this fine-grained level with little to no impact on design performance and area.

design	# used CLBs	# spare CLBs	min period (ns)
MIPS R2000	756	268	185.007
ATR	1876	214	424.542
DES	875	149	166.293

Table 1 Original physical layout statistics

mark size (bits)	800	1568	2592	3200	3872	4608	5408	6272	7200	8192
# ASCII chars	114.29	224.00	370.29	457.14	553.14	658.29	772.57	896.0	1028.6	1170.3
% resources	3.31	6.48	10.71	13.23	16.01	19.05	22.35	25.93	29.76	33.86
% timing	-1.04	-0.47	3.17	-7.15	-4.69	1.65	-11.53	2.47	11.95	-5.23

Table 2 MIPS R2000 - Impact of watermark size on resources and speed

mark size (bits)	32	800	1568	2944	4608	5984	6848
# ASCII chars	4.57	114.29	224.00	420.57	658.29	854.86	978.29
% resources	0.05	1.33	2.61	4.90	7.68	9.97	11.41
% timing	-10.74	3.46	-25.93	-7.99	-13.50	10.25	-1.57

Table 3. ATR - Impact of watermark size on resources and speed

mark size (bits)	32	800	1568	2528	3200	3872	4768
# ASCII chars	4.57	114.29	224.00	361.14	457.14	553.14	681.14
% resources	0.11	2.86	5.60	9.03	11.43	13.83	17.03
% timing	-22.98	-14.83	-5.07	-1.90	11.05	-11.93	-3.28

Table 4. DES - Impact of watermark size on resources and speed

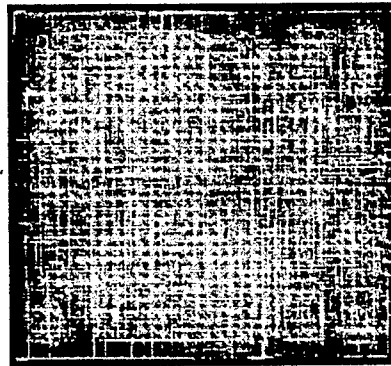
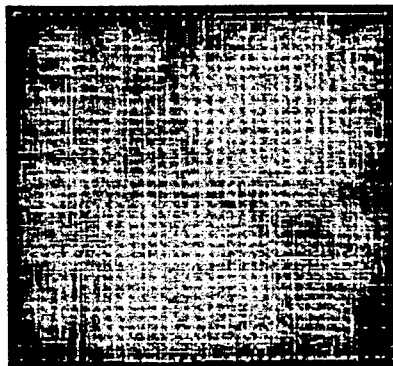


Figure 3a (left). DES original layout
Figure 3b (right). DES with 4768 bit watermark

Acknowledgements

The authors would like to thank Prof Brad Hutchings and Peter Bellows for their assistance. This work was supported by the Defense Advanced Research Projects Agency of the United States of America, under contract F30602-96-C-0350 and subcontract QS5200 from Sanders, a Lockheed Martin company

References

- Anderson, R, and Kuhn, M Tamper resistance - A cautionary note *Proceedings of the Second USENIX Workshop on Electronic Commerce* (1996), 1-11
- Charbon, E Hierarchical watermarking in IC design *Proceedings of the Custom Integrated Circuits Conference '98* (1998)
- Cox, IJ et al Secure spread spectrum watermarking for images, audio and video *Proceedings of the Third International Conference on Image Processing* (1996), 243-246
- Diffie, W and Hellman, M New directions on cryptography *IEEE Transactions on Information Theory* IT-22, 6 (Nov 1976), 644-654
- Hong, I, and Potkonjak, M Behavioral synthesis techniques for intellectual property protection unpublished manuscript (1997)
- Kahng, A B et al Watermarking techniques for intellectual property protection *Proceedings of the Design Automation Conference '98* (1998)
- Leonard, J and Mangione-Smith, W H A case study of partially evaluated hardware circuits: Key-Specific DES *Field Programmable Logic* London, England (1997)
- Schneier, B *Applied Cryptography - Protocols, Algorithms, and Source Code in C* New York: John Wiley & Sons (1996)
- Swanson, M D et al Transparent robust image watermarking *International Conference on Image Processing* (1996), 211-214
- Trimberger, S Personal communication Xilinx Corporation (1997)
- Villasenor, J et al Configurable computing solutions for automatic target recognition *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* Ed Arnold, J and Pocek, K L Napa, CA (1996), 70-79
- Wolfgang, R B and Delp, E J A watermark for digital images *Applications of Toral Automorphisms* 3 (1996), 219-222
- Xilinx *The Programmable Logic Data Book* San Jose, CA (1996)

Watermarking Techniques for Intellectual Property Protection*

A B Kahng, J Lach[†], W H Mangione-Smith[†], S Mantik, I L Markov,
M Potkonjak, P Tucker[‡], H Wang and G Wolfe

UCLA Computer Science Dept, Los Angeles, CA 90095-1596

([†]) UCLA Electrical Engineering Dept, Los Angeles, CA 90095-1594

([‡]) UCSD Computer Science & Engineering Dept, La Jolla, CA 92093-0114

Abstract

Digital system designs are the product of valuable effort and know-how. Their embodiments, from software and HDL program down to device-level netlist and mask data, represent carefully guarded intellectual property (IP). Hence, design methodologies based on IP reuse require new mechanisms to protect the rights of IP producers and owners. This paper establishes principles of *watermarking-based* IP protection, where a *watermark* is a mechanism for identification that is (i) nearly invisible to human and machine inspection, (ii) difficult to remove, and (iii) permanently embedded as an integral part of the design. We survey related work in cryptography and design methodology, then develop desiderata, metrics and example approaches – centering on *constraint-based* techniques – for watermarking at various stages of the VLSI design process.

1 Introduction

The advance of processing technology has led to a rapid increase in IC design complexity. The economic drivers are compelling: only by putting more integration and more function on a single die, and by achieving more revenue per wafer, can multi-billion dollar foundries be amortized over their useful lifespan. At the same time, market forces have led to more design starts, shorter design cycle times and greater time-to-market pressures. Industry organizations have documented a compounding “design productivity shortfall” [26], which demands ever-larger design teams with each successive process generation just to maintain a given level of design competitiveness.¹ Finally, system design costs are increasingly impacted by software, which accounts for up to 70% of total development cost in recent design projects.

In response to these trends, *reuse-based* design methodologies for both hardware and software have been embraced as a means of achieving design productivity on par with the underlying silicon technology. The reuse-based vision is predicated on easily accessible, easily integrable “virtual components”. Pure IP companies, third-party ASIC libraries, tools for IP integration, and industry or-

* Work by M. Potkonjak and G. Wolfe supported in part by DARPA under grant N66001-97-2-8901. Work by W. H. Mangione-Smith and J. Lach supported by the Defense Advanced Research Projects Agency of the United States of America, under contract DAB763-95-C-0102 and subcontract QS5200 from Sanders, a Lockheed Martin company. Work by A. B. Kahng, S. Mantik, I. L. Markov, P. Tucker and H. Wang supported by a grant from Cadence Design Systems, Inc.

¹ According to [26] the available transistor density has increased by 58%/year over the last 20 years; designer efficiency (measured in transistors designed per staff-month) has increased by only 21%/year over the same period.

ganizations such as the VSI Alliance [33] have created high expectations for the value and reusability of design IP. Nonetheless, a recognized obstacle to reuse-based methodologies is the lack of mechanisms to protect the rights of IP creators and owners.²

From both the research and implementation points of view, *intellectual property protection* (IPP) poses a unique set of new requirements that must be addressed by mathematically sound, yet practical, techniques. In this work, we establish principles for development of new *watermarking-based* IPP procedures. These principles, which are centered around the use of *constraints* to “sign” the output of a given design synthesis or optimization, are compatible with current IP development tools infrastructure. Furthermore, our principles apply to the protection of both hardware and software (e.g., Verilog or C++ code) IP.

A Motivating Example: 3SAT

We illustrate key ideas behind watermarking-based IPP using the *satisfiability* (SAT) problem [10]:

SAT (U, C)

Instance: A finite set of variables U and a collection $C = \{c_1, c_2, \dots, c_m\}$ of clauses over U .

Question: Is there a truth assignment for U that satisfies all the clauses in C ?

For example, $U = \{u_1, u_2\}$ and $C = \{\{u_1, u_2\}, \{\bar{u}_1\}, \{\bar{u}_1, \bar{u}_2\}\}$ is a SAT instance for which the answer is positive (a satisfying truth assignment is $t(u_1) = F$ and $t(u_2) = T$). On the other hand, if we have collection $C' = \{\{\bar{u}_1, u_2\}, \{\bar{u}_1, \bar{u}_2\}, \{u_1\}\}$, the answer is negative. SAT is well-known as the first problem shown to be NP-complete, and the starting point for establishing the known body of NP-completeness results [10]. Problems from many application domains have been modeled as SAT instances. In VLSI CAD, SAT formulations have been used in testing [4, 8, 11, 20], logic synthesis and physical design [8].

We now illustrate the *constraint-based watermarking* of a SAT solution. For convenience, we assume the 3SAT restriction of the problem where each clause has exactly three variables. Consider the following 3SAT instance:

$$U = \{u_1, u_2, \dots, u_{14}\}$$

$$C = \{\{\bar{u}_1 \bar{u}_2 u_9\}, \{\bar{u}_1 \bar{u}_3 \bar{u}_4\}, \{\bar{u}_1 u_2 \bar{u}_5\}, \{u_1 \bar{u}_2 u_{10}\}, \{\bar{u}_1 \bar{u}_3 u_8\}, \{\bar{u}_1 \bar{u}_3 u_7\}, \\ \{u_1 \bar{u}_5 u_7\}, \{\bar{u}_1 \bar{u}_6 \bar{u}_{12}\}, \{\bar{u}_1 u_{10} u_{12}\}, \{\bar{u}_1 u_6 u_9\}, \{\bar{u}_2 \bar{u}_3 \bar{u}_{10}\}, \{u_2 \bar{u}_5 \bar{u}_{14}\}, \\ \{\bar{u}_2 u_7 u_8\}, \{u_2 \bar{u}_8 u_9\}, \{u_3 u_4 u_8\}, \{u_3 u_5 \bar{u}_7\}, \{u_3 u_8 u_{13}\}, \{u_3 \bar{u}_9 \bar{u}_{11}\}, \\ \{u_3 u_{10} \bar{u}_{12}\}, \{\bar{u}_4 \bar{u}_7 \bar{u}_8\}, \{\bar{u}_5 \bar{u}_8 \bar{u}_{12}\}, \{u_4 \bar{u}_7 u_{13}\}, \{\bar{u}_5 \bar{u}_9 \bar{u}_{11}\}, \{\bar{u}_5 u_7 u_9\}, \\ \{u_6 u_{10} u_{11}\}, \{u_6 \bar{u}_8 \bar{u}_{12}\}, \{u_7 u_9 \bar{u}_{12}\}, \{u_7 u_9 \bar{u}_{13}\}, \{u_9 u_{11} \bar{u}_{14}\}, \{u_{10} u_{11} \bar{u}_{12}\}\}$$

Our goal is to alter the given 3SAT instance such that (i) any satisfying assignment (“solution”) to the modified instance is a solution to the original instance, and (ii) both the modified instance and the solution contain information (i.e., a “signature”) that uniquely identifies the author of the solution.

²For example, the Virtual Socket Interface (VSI) Alliance has identified six key technologies that must be in place to enable industrial-strength virtual component-based synthesis. In addition to system verification, mixed-signal design integration on-chip bus manufacturing-related test and system-level design, intellectual property protection is considered to be a crucial enabling technology [29].

Enumeration of the solution space indicates that the given 3SAT instance has 556 different satisfying assignments. We impose additional *constraints* in the form of extra 3-literal clauses, using the simple (case-insensitive) encoding $A - u_1, B - \bar{u}_1, C - u_2, D - \bar{u}_2, \dots, Y - u_{13}, Z - \bar{u}_{13}, \text{space} - u_{14}$ to encode a signature.³ We choose the signature "Watermarking Techniques for Intellectual Property Protection University of California at Los Angeles VLSI CAD LAB", which adds 38 new clauses to the instance. After adding these constraints to the initial instance, the number of satisfying assignments decreases to 2. We claim that any satisfying assignment for this augmented 3SAT instance contains our signature, and that the likelihood of someone else generating such a solution by chance is 2 in 556, or 0.00496. In this example, the addition of a watermark incurs no overhead; it simply guides which solution is selected. The watermarking strategy is also *non-intrusive* (more specifically, it is based on *pre-processing* of the input instance), in that any existing solution strategy remains applicable to the augmented (watermarked) 3SAT instance.⁴ In our experience, many commonly encountered NP-complete formulations can also be watermarked using similar constraints.

1.1 Organization of the Paper

The remainder of this paper is organized as follows. Related concepts in artifact watermarking, cryptography and IP-based synthesis are surveyed in Section 2. Principles and desiderata (e.g., protection requirements) of non-intrusive, constraint-based IP watermarking are discussed in Section 3. A review of cryptography background and supporting tools (e.g., one-way functions, pseudo-random encrypted streams, digital signatures, and strength analyses) is given in Section 4. Section 5, as well as the above discussion in Section 1, suggests that non-intrusive (pre- or post-processing based) IP watermarking with constraints is often easy to implement with no significant added complexity or loss of solution quality. While an analysis of typical attacks in Section 4 shows that not all possible protection levels can be achieved with known algorithms, we nonetheless conclude in Section 6 that constraint-based watermarking has significant potential to protect IP and support design reuse.

2 Related Work

We now survey related work in watermarking-based IPP, cryptography, and IP-based synthesis.

2.1 Artifact vs. IP Protection Watermarking

A *watermark* is a mark that is (i) embedded into an artifact (text, image, video, audio) or piece of intellectual property (hardware, software, algorithm, data organization), (ii) designed to identify the author, the source, the used tools and techniques and/or recipient of the artifact or the intellectual property, and (iii) difficult to detect and remove. It is important to distinguish traditional requirements for *artifact watermarking* from those governing the *IP protection* applications that we address. Artifact watermarking simply adds a signature into a given artifact *without regard to maintaining correctness or function*. "Transparency" of the signature stems from imperfections in human auditory and visual systems: the artifact (e.g., a digitized photograph) is actually changed, but the human eye cannot perceive the change.⁵

In contrast, watermarking for IP protection imposes much stronger constraints because the watermarked IP must remain *functionally correct*. For example, one cannot arbitrarily introduce extra lines of code into a Verilog program, or extra devices and inter-

connects into a transistor-level layout. Our discussion is centered around the following key idea: watermarking for IPP is most practically accomplished by imposing a set of additional *constraints* during the design and implementation of IP, so as to uniquely encode the signature of the author. Since 1996, the effectiveness of this generic scheme for watermarking-based IPP has been demonstrated at the level of algorithms [15], behavior [14], logic synthesis and physical design [16], as well as in FPGA designs [18, 19].

2.2 Cryptography

Modern age cryptography grew from the seminal work of Diffie and Hellman [9], who introduced public-key cryptography based on the computational intractability of certain mathematical tasks. Since 1976, cryptographic algorithms and techniques have evolved through vigorous innovation and public scrutiny, resulting in a variety of digital signature mechanisms, as well as protocols for secret splitting, timestamping, proxy signatures, group signatures, key escrow, oblivious transfer, oblivious signatures, digital cash, etc. [27, 22]. The link between cryptography and watermarking is fundamental: cryptography provides the theoretical foundations as well as the algorithmic and protocol infrastructure that support watermarking-based IPP and provide a wide spectrum of authorship protection services.

2.3 IP-Based Synthesis

As noted above, short design times, increased device counts and design starts, and foundry amortization have together forced a change in design methodology. The new semiconductor business regime is based on IP reuse. No other regime is compatible with rapid turnaround and high device counts; no other regime enables ASIC vendors to keep their foundries full of high-value product.⁶

Less than two years ago, the VSI Alliance and CFI Component Information Library Project were first announced. Today, at least three major industry organizations – RAPID (IP providers) [32], SI² [34] (ASIC vendors), and VSIA [33] (a large organization of EDA vendors, ASIC vendors, system houses and IP providers) – are actively building the industry infrastructure for IP-based design.⁷ Several missing infrastructure pieces are technical, with deep implications for the associated EDA technology and design methodologies.⁸ Other missing pieces include the standards for representing design IP. However, arguably the most pressing infrastructure issues are legal: what are the risks faced by ASIC suppliers and EDA tools vendors as they incorporate third-party IP? Who holds accountability for design success? How will the rights of IP creators and owners be protected? It is notable that despite their varying perspectives, each of the three major industry organizations has a working group for legal issues.

3 Precepts and an Approach for Constraint-Based IPP

In this section we develop basic precepts, and a general constraint-based approach, for watermarking IP protection. Our discussion will abstract the design process as a form of optimization, and we will focus on opportunities for non-intrusive watermarking (i.e.,

⁶Interestingly, the vision of pervasive IP reuse can be viewed as simply a refinement of earlier visions which saw the inability of the structured-custom methodology to scale with design complexity as a main driver for "methodology convergence".

⁷The early CFI effort spawned the Pinnacles Component Information Standard, and CFI subsequently became SI² (Silicon Integration Initiative).

⁸For example, how reusable IP will be bundled with standardized test and simulation "envelopes", or the form of reusable IP and the manner in which it will "mix and match", remains unclear. Current visions encompass varying degrees of "hardness" of the IP, e.g., soft (HDL program), medium (HDL program + floorplan), hard (GDSII stream file), etc. Harder forms of IP might have greater value since they would embody greater amounts of design effort. At the same time, hard IP is less reusable due to its well-defined shape and inherent timing/noise/thermal context; it also allows less flexibility in floorplanning and routing due to constraints on over-the-block routing (e.g., timing and signal integrity margins). It remains to be seen how "parameterizable" an IP block can be in terms of area-time tradeoffs, migration to alternate processes, routing resource utilization, etc.

³For example, the signature "cat dog fox" would be encoded using the extra clauses $\{\{u_2, u_1, \bar{u}_{10}\}, \{u_{14}, \bar{u}_2, u_9\}, \{u_4, u_{14}, \bar{u}_3\}, \{u_8, \bar{u}_{12}, u_{14}\}\}$ (we pad the end of the message with an extra space to maintain three literals per clause).

⁴This observation holds for the three major classes of SAT heuristics: (i) random search [25, 7], (ii) nonlinear programming relaxation and rounding [12], and (iii) a variety of BDD-based techniques [3].

⁵Artifact watermarking has been used for thousands of years. Only with the proliferation of digital media has it attracted wide research and economic interest, e.g., for protection of audio [1, 15], text [21, 2], image [5], and video.

methods that can be transparently integrated within existing design flows via pre- or post-processing)

3.1 Context for Watermarking

The following ingredients form the *context* for a non-intrusive watermarking procedure:

- An *optimization problem* with known difficult complexity, corresponding to some design synthesis task. By difficult, we mean that either achieving an acceptable solution, or enumerating enough acceptable solutions, is prohibitively costly. The solution space of the optimization problem should be large enough to accommodate a digital watermark.
- A well-defined *interpretation* of the solutions of the optimization problem as intellectual property.
- Existing *algorithms* and/or *off-the-shelf software* that solve the *optimization problem*, likely without any kind of watermarking involved. Typically, the “black box” software model is appropriate, and is moreover compatible with defining the watermarking procedure by composition with pre- and post-processing stages.⁹
- *Protection requirements* that are largely similar to well-understood protection requirements for currency watermarking. Examples of such requirements, which are discussed in Section 4 below, include: (i) removing or forging a watermark must be as hard as re-creating the design; and (ii) tampering with a watermark must be provable in court.

A non-intrusive watermarking procedure then applies to any given instance of the optimization problem, and can be attached to any specific algorithms and/or software solving it. Such a procedure can be described by the following components:

- A *use model* or *protocols* for the watermarking procedure. This is not the same as algorithm descriptions; it is less formal, and can be helpful in revealing possible attacks beyond the generic types noted above. For example, algorithms assume a cell numbering, while renumbering cells can defeat a watermarking procedure (something that can be seen only at the protocol level). In general, each watermarking scheme must be aware of attacks based on design symmetries, renaming, reordering, small perturbations (which may set requirements for the structure of the solution space), etc.
- Algorithmic descriptions of the *pre-* and *post-processing* steps of the watermarking procedure.
- *Strength and feasibility analyses* showing that the procedure satisfies given protection requirements on a given instance. Strength analysis requires metrics, as well as structural understanding of the solution space (e.g., “barriers” (with respect to local search) between acceptable solutions). Feasibility analysis requires measures of solution quality, whether a watermarked solution remains well-formed, etc.
- *General robustness analyses*, including discussion of susceptibility to typical attacks, discussion of possible new attacks, performance guarantees (including complexity analysis) and implementation feasibility.

Before describing a general strategy for embedding digital watermarks, we observe that optimization problems with known watermarking procedures share several common features: (i) having

⁹Watermarking the results of non-deterministic and/or unknown algorithms – or even “hand-made” results – is possible as well. IP protection can even be achieved to some extent with black-box off-the-shelf software that is viewed as a one-way function mapping inputs to design solutions. In this discussion we focus only on the simple model involving known deterministic algorithms.

multiple acceptable solutions (we typically accept suboptimal solutions for NP-hard problems), (ii) solved by optimization heuristics; and (iii) discrete in nature.¹⁰

3.2 General Strategy for Constraint-Based IPP

Our general strategy is to map an author’s signature into a set of constraints (“desired relations”) which can independently hold for a particular solution (or independence can be assumed, via some manipulations). If disproportionately many of these constraints are satisfied the presence of the signature is indicated, and vice versa. Choosing the type of constraints, and the tactic (e.g., pre- or post-processing) by which we make it likely for more of them to be satisfied than would otherwise be expected, is what instantiates a particular watermarking algorithm from the general strategy. These choices can dramatically affect the strength of the watermark and the degradation of solution quality caused by watermarking. To facilitate later discussion, we now describe generic watermarking and signature verification procedures using “Alice (and Bob)” scenarios, where Alice uses watermarking to protect some IP (below, Bob will attempt to subvert such protection).

Generic Watermarking Procedure. Alice wishes to protect some IP that involves many stages of processing. She chooses to watermark one or more of these stages. The results of these stages now carry a watermark which will propagate down to the output of further stages all the way down to the final result. Clearly the amount of watermarking she imposes on a particular stage trades off with the degree of degradation of quality of the final result. Alice watermarks each stage by selecting a set of “constraints”, then using preprocessing of the stage’s input and postprocessing of the stage’s output to encourage a disproportionate number of these constraints to be satisfied. Note that Alice need not tell anyone which constraints correspond to her signature.

Generic Signature Verification Procedure. To demonstrate that a particular stage was watermarked Alice must show that its solution (which may have been passed on undisturbed to other stages and perhaps all the way to the final result) satisfies a disproportionate number of her watermarking constraints. By identifying the watermarking constraints, determining how many of them are satisfied, and calculating P_c – the probability of so many (or more) of the constraints being satisfied by coincidence – Alice can verify that her signature is present. A strong proof of authorship corresponds to a low value for P_c . Note that to show this to other people, Alice must reveal her signature and, hence, the chosen constraints.

4 Analysis of Constraint-Based Watermarking

We now analyze the constraint-based watermarking strategy and present the cryptographic background and support tools that are necessary for the analysis. We first describe how to map a signature into a set of constraints and the method by which we determine the strength of the watermark in a watermarked solution. We then discuss typical forms of attack on our scheme and the obstacles that prevent these attacks from succeeding.

4.1 Selection of Constraints

Given a pseudorandom number generator and a particular type of constraint it is a simple matter to select a set of X constraints where each one is determined independently. It is only slightly more work to select a set of X constraints with no constraint repeated. Thus, the task of mapping an author’s signature into a set of constraints can be reduced to the task of seeding a pseudorandom number generator with the signature. This is also easy. Suppose that the author’s signature is a particular text message. We can convert this message into a cryptographically sound pseudorandom bit stream by simply hashing the message (using a one-way hash function such as MD5 [24]) and using the hash as a seed for a stream cipher such as RC4 [35].

¹⁰We believe continuous watermarking is possible as well, e.g., by mapping into discrete watermarking by Fourier transform. However, watermarking has traditionally been of more relevance to discrete problems.

4.2 Proof of Authorship

A watermark's *proof of authorship* is expressed as a single value, P_c , which is the probability of so many (or more) of the selected constraints being satisfied. Essentially, P_c is the probability of a non-watermarked solution carrying our watermark by coincidence. We wish this probability to be convincingly low so as to have a strong proof of authorship. When we cannot compute P_c exactly it is acceptable to overestimate it so that we actually report an upper bound on P_c . Computing such an upper bound on P_c is typically straightforward. Let p be the probability of satisfying a single random constraint by coincidence. This value, or a fairly tight upper bound on it, is usually obvious from the definition of a constraint. Here we assume that p is independent of whether the other constraints were satisfied. Let C be the number of imposed constraints. Let b be the number of these constraints that were *not* satisfied. Let X be a random variable that represents how many of the C constraints were not satisfied. Now P_c can be computed as a sum of binomials, i.e., the probability that coincidentally only b or less of C constraints were not satisfied is given by $P_c \equiv P(X \leq b) = \sum_{i=0}^b \binom{C}{i} p^i (1-p)^{C-i}$. This analysis assumes that p is independent of whether other constraints are satisfied, an assumption that is often untrue. However, when the number of imposed constraints (C) is sufficiently small, we have a very good approximation.

4.3 Typical Attacks

There are several general ways of attacking our watermarking scheme. Here we discuss the more prominent ones: finding "ghost signatures", tampering, and forging. We analyze these attacks using "Alice and Bob" scenarios.

Attack: Finding Ghosts. Bob wishes to steal IP from Alice and claim it as his own. He knows that Alice has protected her IP (i.e., the solution to a particular stage of the design process) with a watermark, but will claim that the IP *also* contains his own watermark. Bob thus attempts to find a *ghost signature*, namely, a signature that corresponds to a set of constraints that yields a favorable P_c , but which was discovered after fact instead of being actually watermarked into the solution. To be convincing, Bob must find a ghost signature that yields a sufficiently convincing value P_c .

Bob has only two approaches. He may choose a set of constraints (presumably ones that yield a good proof of authorship P_c) and then attempt to find a signature that corresponds to this set. This requires reversing the cryptographically secure one-way functions that convert a signature into a set of constraints, which is hard. Alternatively, Bob may try a brute-force approach to find a signature that corresponds to a set of constraints that yields a convincing proof of authorship P_c . However, this brute-force attack becomes computationally infeasible if the threshold for proof of authorship is set sufficiently low (e.g., $P_c \leq 2^{-56}$).

Attack: Tampering. If Bob cannot find a convincing ghost signature, he may decide to *tamper* with Alice's solution. Ideally, such tampering would completely remove Alice's signature and add Bob's own signature. Bob can do this by simply re-solving the problem from scratch with his own watermark encoded, then continuing through subsequent processing stages based upon the output he obtains. Nothing can be done to stop this directly. However, we believe that in realistic scenarios, Bob cannot afford to redo all of the subsequent phases of the design process, particularly if the watermarking occurred relatively early in the process.

There are realistic means by which Bob can tamper with a solution without having to re-solve every subsequent stage of the process. Generally, these amount to transforming the solution output by the last phase of the design process, where the transformation has a similar effect on the output of the watermarked phase of the design process. Specific changes that Bob makes to the final solution will likely correspond to (i) local perturbations of the solution to the watermarked phase, or to (ii) global-scale transformations

such as those which exploit a symmetry of the design representation. Given that Bob is limited to these kinds of tampering attacks, it is critical that Alice's watermarking technique be resistant to such transformations.¹¹

Attack: Forging. Finally, Bob may attempt to subvert Alice's watermark by inappropriately watermarking other solutions with Alice's watermark. In other words, Bob wishes to *forge* Alice's signature. To do this, Bob needs a signature that he can convince others belongs to Alice. If a signature corresponds simply to a text message (as it has so far in this discussion) then Bob's task is easy: he simply chooses a text message resembling one that Alice would use. However, such attacks can be easily prevented by using a public key encryption system [23]. Any message actually signed by Alice would be encrypted with her private key, yet verifiable with her public key. Notice that the private key is not compromised even if messages that encoded with it are compromised, so Alice may still demonstrate the presence of her watermark to anyone who knows her public key, without compromising her private key. Thus, Bob is able to forge a message from Alice only if he knows her private key.

5 IP Watermarking Synthesis Examples

In this section, we sketch three examples of IP watermarking approaches, in three very distinct domains. Our intent is to illustrate the wide-ranging applicability of the principles developed above.

5.1 Preprocessing in System-Level Design

At the system level, instruction and data caches consume a significant portion of the overall area, and often have crucial impact on system timing and power consumption [17]. Much effort has been devoted to allocating minimal cache structures and optimizing code for effective cache utilization [30]. A particularly successful technique is *cache line coloring* [13].

Given a code segment and input data benchmarks, cache line coloring code optimization seeks a permutation of basic block code segments such that the mapping of code to cache entries minimizes the cache miss ratio over the given benchmarks. The problem can be modeled as follows. The program is profiled with respect to the benchmark data, and spatial (frequent sequences of sequentially executed code) and temporal (frequent control sequences) correlations noted among basic blocks of code. The program is modeled using a control data flow graph, where a graph node corresponds to a set of instructions that are encompassed in a single basic block and fit exactly one cache line. Weighted edges between nodes correspond to spatial or temporal correlations that exceed given threshold values (modeling accuracy thus depends on the thresholds for edge inclusion). The problem of minimizing cache misses is equivalent to finding a solution to graph coloring using a given fixed number of colors (corresponding to available cache lines).¹²

To watermark such designs, the initial design constraints may be augmented with additional constraints corresponding to the digital signature of the designer. For example, following the technique for watermarking of graph coloring solutions proposed by Hong and Potkonjak [14], one may add additional edges to the graph according to some encrypted signature of the author. Therefore, the signature will be embedded in the activation path which transfers data between two levels of hierarchy.

¹¹Note that since the attacker does not know which constraints correspond to the author's signature, tampering attacks might not be able to ruin the proof of authorship before they significantly degrade the quality of the final solution (at which point the tampered solution ceases to be useful); see [16] for experience in the physical design realm. However, it seems possible for an attacker to use tampering methods to remove a signature that is known to him, or to add an entirely new signature.

¹²Kirovski et al. [17] have experimentally shown that this optimization results in significant performance increase. In general, such optimizations can play an important role in the design of modern multimedia, communications, or low-power systems-on-silicon.

5.2 Postprocessing in Physical-Level FPGA Design

One method of watermarking an FPGA at the physical level involves manipulating unused portions of the configuration bitstream. Informed parties can then extract the mark from the bitstream. There is no effect on the function of the design during insertion or extraction because only unused portions of the design are altered. This approach can be implemented through pre-processing, iterative, or post-processing techniques. The advantage of post-processing is that it does not impact other CAD design tools, and has zero impact on design performance, area or power consumption. The disadvantage of this approach is that the watermark is not embedded in the functional part of the design; given enough information, the watermark can be removed without affecting design functionality. An example of an iterative approach can be found in the work by Lach et al [18, 19].

An example of a purely post-processing approach involves inserting the watermark into the control bits for unused outputs from configurable logic blocks (CLBs). Certain bits in the configuration bitstream that control multiplexers for the CLB outputs can be replaced by watermark bits if the CLB outputs are not used. For example, the Xilinx 4000 family of FPGAs contain CLBs with four outputs [31]. Two outputs (X and Y) are combinational, while the others (XQ and YQ) can be used in sequential designs. The two combinatorial outputs are each controlled by a 2-to-1 multiplexer, and the two sequential outputs are each controlled by three 2-to-1 multiplexers and one 4-to-1 multiplexer. Figure 1 shows the control layout of the 4000 family's CLB outputs.

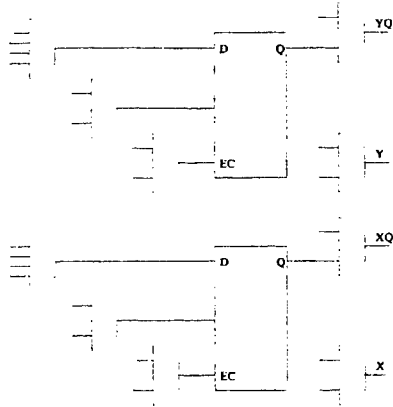


Figure 1: Control directly attributed to CLB outputs

The number of configuration bits associated with a multiplexer is equal to (or greater than) the number of required control bits. Therefore, one and two watermark bits can be inserted at each unused 2-to-1 and 4-to-1 multiplexer respectively. Thus, each unused combinatorial output can store one watermark bit and each unused sequential output can store five watermark bits. The total number of watermark bits that can be inserted in an entirely unused CLB is twelve. Table 1 shows the number of watermark bits that can be inserted into various devices within the 4000 family given certain percentages of unused CLB outputs. The numbers calculated here are for an even number of unused combinatorial and sequential outputs.

The process of watermark insertion in this approach is an entirely post-processing step and requires very little added design effort. The tool methodically scans the bitstream searching for unused outputs by finding CLB output pinwires that do not attach to any external CLB interconnect. Upon the detection of unused outputs, the next bits of the watermark are inserted in place of the corresponding multiplexer configuration bits. The size of the watermark is limited by the number of bits made available by this approach. Extracting the watermark is an almost identical process.

% Outputs Unused	Part/# CLBs				
	4006 /256	4010 /400	4013 /576	4020 /784	4025 /1024
1	30	48	69	94	122
5	153	240	345	470	614
10	307	480	691	940	1228
20	614	960	1382	1881	2457

Table 1: Number of bits available for watermarking

The tool finds unused CLB outputs the same way as was done in insertion and pieces the watermark back together by examining the corresponding multiplexer configuration bits.

This FPGA watermarking approach requires little extra design effort, can store fairly large watermarks, allows for easy mark extraction, and has no overhead in terms of design area or performance. However, because a mark is nonfunctional, it may be removed by reverse engineering a design to a stage in the flow before the mark has been applied. Fortunately, most FPGA vendors will not reveal the specification of their configuration streams, specifically to complicate the task of reverse engineering and thus protect the investment of their customers. For example, the Xilinx XC4000 devices follow a form of Pareto's rule: the first 80% of the configuration information can be determined relatively easily by inspection, the next 18% is much more difficult, etc. The complexity is enhanced by an irregular pattern that is not consistent between rows or columns, as a result of the hierarchical interconnect network. Xilinx does not take any specific actions to make their configurations difficult to reverse engineer. However, they do believe that it is difficult to do in general, and they promise their customers that they will keep the bitstream specification confidential in order to raise the bar for reverse engineering [28].

5.3 Preprocessing in Physical Design

Finally, in the context of physical design, we present a new pre-processing based approach for design watermarking. Our approach exploits the flexibility with which *path-based timing constraints* can be satisfied.

Consider the typical elements of an input instance for timing-driven placement and routing:

- physical floorplan, library of physical cell masters, and cell-level netlist
- cell-level performance macromodels for each cell master (e.g., non-linear table models (Synopsys lib, Cadence ctf, OVI ALF, etc.)) for timing and power dissipation analysis
- technology file (models of interconnect RC characteristics, design rules, etc.)
- constraints, which are chiefly (i) "direct" placement and routing constraints (e.g., region-based location constraints arising from the floorplanner, and transmitted in PDEF format), and (ii) performance constraints (e.g., SDF latch-to-latch path timing upper and lower bounds, with false path and multi-cycle constraints specially annotated)

We watermark a design by selecting path timing constraints and replacing them with "subpath" timing constraints. Suppose that we have the path timing constraint $t(C_1 - C_2 - C_3 - \dots - C_{10}) \leq 50ns$ ($C_i \equiv$ cells). We can allocate the timing bound between two sub-paths and replace this constraint by two constraints $t(C_1 - \dots - C_5) \leq 20ns$ and $t(C_5 - \dots - C_{10}) \leq 30ns$. All else being equal, the chance that satisfying the original constraint happens to satisfy both of these subpath constraints is at most one-half.¹³ Constraining on

¹³Note that the allocation would be done with respect to available slack on the path e.g., path delay upper bound minus sum of "intrinsic" cell delays. Also note that constraint satisfaction will likely be determined in the context of final layout.

the order of hundreds of timing paths (from the several millions one finds in typical verbose SDF specifications) is transparent to timing-driven design tools, yet affords strong proofs of authorship. Similar techniques can be applied in the regime of compact SDF timing constraints, or at the budgeting stages of timing-driven design.¹⁴

6 Conclusions

Motivations and antecedents for watermarking-based protection of hardware and software design IP arise in reuse-centric system design, artifact watermarking, and cryptography. In this paper, we have described fundamental precepts, a canonical technique, and example applications for watermarking-based IPP. Several key ideas are as follows:

- Stages of the (hardware, software) design process can typically be viewed as (difficult) *optimization instances* whose solutions constitute intellectual property to be protected
- IP watermarking can typically be achieved by adding *constraints* (e.g., interpreted from a cryptographically secure encoding of the IP owner's signature) to any given design optimization instance
- The addition of constraints can typically be achieved using *pre- or post-processing* of the inputs and outputs, respectively, for a given design optimization. In this way, the watermarking is often transparent to existing algorithms and tools, i.e., it is *non-intrusive*.

We have also noted other aspects of the watermarking context, e.g., protection requirements against typical forms of attack, and cryptography background (one-way functions, cipher streams, and digital signatures). Problem formulations from several domains (high-level design, FPGA design, physical design, as well as SAT) illustrate the general applicability of our techniques, and suggest that non-intrusive IP watermarking with constraints can typically be implemented with no significant added complexity or loss of solution quality. Thus, constraint-based watermarking appears to have significant potential to protect IP and support design reuse.

Our ongoing work develops watermarking-based IPP techniques for many other domains, with particular attention to robustness under various attacks. We also address a number of variant requirements, including fingerprinting, copy detection, and proportionate watermarking (e.g., of hierarchical designs).

References

- [1] W. Bender, D. Gruhl, N. Morimoto and A. Lu, 'Techniques for Data Hiding', *IBM Systems Journal*, 35(3-4) 1996 pp 313-336
- [2] J. T. Brassil, S. Low, N. F. Maxemchuk and L. O'Gorman, 'Electronic Marking and Identification Techniques to Discourage Document Copying', *IEEE Journal on Selected Areas in Communications*, 13(4) (1995) pp 1495-1504
- [3] R. E. Bryant, 'Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification', *Proc. of the International Conference on Computer-Aided Design*, 1995, pp 236-243
- [4] S. T. Chakradhar, V. D. Agrawal and S. G. Rothweiler, 'A Transitive Closure Algorithm for Test Generation', *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(7) (1993) pp 1015-28
- [5] I. J. Cox and M. L. Miller, 'A Review of Watermarking and the Importance of Perceptual Modeling', *SPIE Conf. on Human Vision and Electronic Imaging II*, 3016(1997), pp 92-99
- [6] I. J. Cox, J. Kilian, F. T. Leighton and T. Shamoon, 'Secure Spread Spectrum Watermarking for Multimedia', *Transactions on Image Processing*, 6(12) (1997) pp 1673-87
- [7] M. Davis and H. Putnam, 'A Computing Procedure for Quantification Theory', *Journal of the ACM*, 7(3) (1960) pp 201-215
- [8] S. Devadas, 'Optimal Layout Via Boolean Satisfiability', *IEEE International Conference on Computer-Aided Design*, 1989, pp 294-7
- [9] W. Diffie and M. Hellman, 'New Directions in Cryptography', *IEEE Transactions on Information Theory*, IT-22(6), 1976 pp 644-654
- [10] M. E. Garey and D. S. Johnson, *Computers and Intractability: a Guide to the Theory of NP-Completeness*, Freeman, 1979
- [11] J. Giraldi and M. L. Bushnell, 'Search State Equivalence for Redundancy Identification and Test Generation', *Proc. Intl. Test Conf.*, 1991 pp 184-193
- [12] M. X. Goemans and D. P. Williamson, 'Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming', *Journal of the ACM*, 42(6) (1995), pp 1115-45
- [13] A. H. Hashemi, D. R. Kaeli and B. Calder, 'Efficient Procedure Mapping Using Cache Line Coloring', *SIGPLAN Notices*, 32(5) (1997) pp 171-182
- [14] J. Hong and M. Potkonjak, 'Behavioral Synthesis Techniques for Intellectual Property Protection', unpublished manuscript, 1997
- [15] L. Honey, A. H. Tewfik and K. N. Hamdy, 'Digital Watermarks for Audio Signals', *Proc. of the International Conference on Multimedia Computing and Systems*, 1998 pp 473-480
- [16] A. B. Kahng, S. Mantik, I. L. Markov, M. Potkonjak, P. Tucker, H. Wang and G. Wolfe, 'Robust IP Watermarking Methodologies for Physical Design', *Proc. ACM/IEEE Design Automation Conf.*, 1998
- [17] D. Kirovski and M. Potkonjak, 'System-Level Synthesis of Low-Power Hard Real-Time Systems', *Proc. ACM/IEEE Design Automation Conference*, 1997 pp 697-702
- [18] J. Lach, W. H. Mangione-Smith and M. Potkonjak, 'Fingerprinting Digital Circuits on Programmable Hardware', *Proc. Workshop on Information Hiding*, 1998
- [19] J. Lach, W. H. Mangione-Smith and M. Potkonjak, 'FPGA Fingerprinting Techniques for Protecting Intellectual Property', *Proc. CICC*, 1998
- [20] T. Larrabee, 'Test Pattern Generation Using Boolean Satisfiability', *Proc. International Test Conf.*, 11(1) (1992), pp 4-15
- [21] S. H. Low, N. F. Maxemchuk, J. T. Brassil and L. O. Gorman, 'Document Marking and Identification Using both Line and Word Shifting', *Proc. INFOCOM*, 1995 pp 853-60
- [22] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone, *Handbook of Applied Cryptography*, Boca Raton: CRC Press, 1997
- [23] Pretty Good (tm) Privacy, Phil Zimmerman, Users Guide: Vol 1, Vol 2 and Internal Data Structures from Phil's Pretty Good Software copy: <http://www.atp.11n1.gov/atp/papers/HRH/references/pgp-refs.html>
- [24] R. L. Rivest, 'RFC 1321: the MD5 Message-Digest Algorithm', *Internet Activities Board*, April 1992
- [25] B. Selman, 'Stochastic Search and Phase Transitions: AI Meets Physics', *IJCAI*, 1 (1995), pp 998-1002
- [26] Semiconductor Industry Association, *National Technology Roadmap for Semiconductors*, revised November 1997
- [27] D. R. Stinson, *Cryptography: Theory and Practice*, Boca Raton: CRC Press, 1995
- [28] S. Trimberger, *personal communication*, 1997
- [29] VSI Alliance, 'Fall Worldwide Member Meeting: a Year of Achievement', Santa Clara, CA, October 1997
- [30] H. Wang, T. Sun, Q. Yang, 'Minimizing Area Cost of on-Chip Cache Memories by Caching Address Tags', *IEEE Transactions on Computers*, 46(11) (1997), pp 1187-1201
- [31] Xilinx, *The Programmable Logic Data Book*, Xilinx Corporation, San Jose, 1996
- [32] <http://www.rapid.org>
- [33] <http://www.vsi.org>
- [34] <http://www.sii2.org>
- [35] <http://dcsex.ac.uk/~aba/rsa/rc4.html>

¹⁴In general, the physical design context presents a rich environment for constraint-based watermarking. For example, the physical library information and/or design rules allow variant pin access models for a cell, which will constrain how interconnects attach to pins; extra blockage geometries in cell instances or masters can also be used to constrain the routing; and via and stub rules can again encode a signature within the output of a constraint-driven router [16]. Simple parity-based schemes abound, e.g., based on mirroring of cells, parity of row indices to which cells are assigned [16], routing of wires to the left or right of shield wires, etc. Even performance macromodels (nonlinear table models for timing and power) can be perturbed (thus constraining the performance-driven layout) to influence the layout tool's output.

Fingerprinting Digital Circuits on Programmable Hardware

John Lach¹, William H. Mangione-Smith¹, Miodrag Potkonjak²
University of California, Los Angeles

Department of Electrical Engineering¹
56-125B Engineering IV
Los Angeles, CA 90095
{jlach, billms}@icsl.ucla.edu

Department of Computer Science²
4532K Boelter Hall
Los Angeles, CA 90095
{miodrag}@cs.ucla.edu

Abstract. Advanced CAD tools and high-density VLSI technologies have combined to create a new market for reusable digital designs. The economic viability of the new core-based design paradigm is pending on the development of techniques for intellectual property protection. A design watermark is a permanent identification code that is difficult to detect and remove, is an integral part of the design, and has only nominal impact on performances and cost of design.

Field Programmable Gate Arrays (FPGAs) present a particularly interesting set of problems and opportunities, because of their flexibility. We propose the first technique that leverages the unique characteristics of FPGAs to protect commercial investment in intellectual property through fingerprinting. A hidden encrypted message is embedded into the physical layout of a digital circuit when it is mapped into the FPGA. This message uniquely identifies both the circuit origin and original circuit recipient, yet is difficult to detect and/or remove. While this approach imposes additional constraints on the back-end CAD tools for circuit place and route, experiments involving a number of industrial-strength designs indicate that the performance impact is minimal.

1 Introduction

We introduce a fingerprinting technique that applies cryptographically encoded marks to Field Programmable Gate Array (FPGA) digital designs in order to support identification of the design origin and the original recipient (i.e. customer of record). The approach is shown to be capable of encoding long messages and to be secure against malicious collusion. Nonetheless, the technique is efficient and requires low overhead in terms of hardware area and circuit performance.

1.1 Motivation

It is generally agreed that the most significant problem facing digital IC designers today is system complexity. The process of large system implementation has followed an evolutionary path from multiple ICs, through single ICs, and now into portions of ICs. For example, twenty years ago a 32-bit processor would require several ICs, ten years ago a single IC was necessary, and today a 32-bit RISC core requires approximately 25% of the StrongARM 110 device developed by Digital Semiconductor in collaboration with ARM Limited [1-3]. Fortunately, complex

systems tend to be assembled using smaller components in order to reduce complexity as well as to take advantage of localized data and control flows. This trend toward partitioning enables design reuse, which is essential to reducing development cost and risk while also shortening design time. While systems designers have employed design reuse for years, what is new is that the boundaries for component partitions have moved inside of the IC packages.

A number of design houses have appeared that provide a wide range of modules, such as parameterized memory systems, I/O channels, ALUs and complete processor cores. These reusable modules are collectively known as Intellectual Property (IP), as they represent the commercial investment of the originating company but do not have a natural physical manifestation.

Direct theft is a concern of IP vendors. It may be possible for customers, or a third party, to sell an IP block as their own without even reverse engineering the design. Because IP blocks are designed to be modular and integrated with other system components, the thief does not need to understand either the architecture or implementation.

This paper presents a deterrent to such direct misappropriation. The essential idea involves embedding a digital mark, which uniquely identifies the design origin and recipient, in an IP block. This mark (origin signature + recipient fingerprint) allows the IP owner to not only verify the physical layout as their property but to identify the source of misappropriation, in a way that is likely to be much more compelling than the existing option of verifying the design against a registered database. This capability is achieved with very low overhead and effort and is secure against multiparty collusion.

Any effective fingerprinting scheme should achieve the following goals:

1. The mark must be difficult to remove.
2. It must be difficult to add a mark after releasing the IP to a customer.
3. The mark should be transparent.
4. The mark should have low area and timing overhead and little design effort.

The benefits of properties 1 and 2 are readily apparent and can be achieved by integrating the mark into the design. It then becomes more difficult to detect and remove, as there is no clear distinction between the mark and parts necessary to the design, thus making it more difficult to add another mark. Any attempts to remove the mark or add another incur a much greater risk of changing the design function.

Property 3 is important to provide IP protection across a wide community of developers and is the key to extending the benefits of IP protection to those who do not employ fingerprinting. By masking the presence of a mark, we discourage all forms of theft. Ayres and Levitt compared the impact of obtrusive and unobtrusive theft-preventive measures for automobiles [4]. They provide compelling evidence that unobtrusive tracking measures are significantly more effective at reducing theft than measures that are apparent to the thief, because of the deterring impact of uncertainty. We believe that a parallel exists between measures such as fingerprinting and more apparent techniques such as conventional design encryption.

Property 4 requires that the overhead in terms of area, timing, and design effort needed to mark the design is not excessive.

1.2 Motivational Example

Our fingerprinting approach builds upon two existing techniques: an FPGA watermarking technique that hides a mark [5] and an FPGA design tiling and partitioning technique that greatly reduces the cost of generating many different circuit instances which are functionally equivalent [6].

FPGAs are programmable logic devices that are composed of an array of configurable logic blocks (CLBs) which are connected via a programmable network. A device is configured with a bitstream generated by CAD tools specifying the functionality of the CLBs and the routing of the network. While the concepts developed here can be applied to a wide range of FPGA architectures, all of the discussion and experimental work will be conducted in the context of the Xilinx XC4000 architecture [7]. CLBs in an XC4000 each contain two flip-flops and two 16x1 lookup tables (LUTs). A hierarchical and segmented network is used to connect CLBs in order to form a specific circuit configuration.

A secure and transparent mark can be placed in an FPGA design using a previously developed FPGA watermarking technique [5]. Consider the case of PREP Benchmark #4 [8], a large state machine, which can be mapped into a block of 27 CLBs. This mapping results in 3 unused CLBs, or $3 \times 32 = 96$ unused LUT bits. Each unused LUT bit is used to encode one bit of the mark. Figure 1 shows the layout of the original design as produced by the standard Xilinx backend tools, while Figure 2 shows the layout for the same design after applying the mark constraints to the three unused CLBs and re-mapping the design. The marked CLBs are incorporated into the design with unused network connections and neighboring CLB inputs, further hiding the mark.

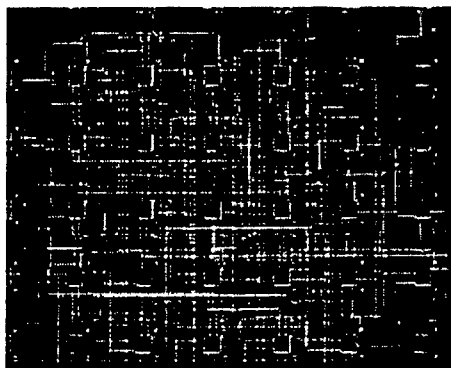


Figure 1. Original layout of PREP benchmark #4



Figure 2. Marked layout of PREP benchmark #4 with a 96-bit mark

Tiling [6] is then used to efficiently support fingerprinting. Consider the Boolean function $Y = (A \wedge B) \wedge (C \vee D)$, which might be implemented in a tile containing four CLBs as shown in Figure 3.1. This configuration contains one spare CLB, and its LUT can hold a mark indicating the owner's signature and recipient's fingerprint. Each recipient could receive this original configuration with a unique fingerprint. Using the same base configuration for a different recipient, and therefore a different fingerprint, would facilitate simple comparison collusion (e.g. XOR), as the only

difference between the designs would be the fingerprint. Note however that each implementation in Figure 3.I-IV is interchangeable with the original, as the interface between the tile and the surrounding areas of the design is fixed and the function remains unchanged. The timing of the circuit may vary, however, due to the changes in routing. With several different instances of the same design, comparison collusion would highlight functional differences, thus disguising the differences between the various recipients' fingerprints.

If a design had four tiles the size of the instance in Figure 3 (i.e. the design is a 2x2 array of tiles; each tile is a 2x2 array of CLBs), the total number of CLBs in the design would be 16. Assuming each CLB has a total of 32 LUT bits and each tile has one CLB free, 128 LUT bits would be available to encode a mark. Each tile has four instances, making the total number of design instances $4^4=256$. A non-tiled design would have $\binom{16}{4} = 1820$ possible instances, but each instance requires a complete

execution of the backend CAD software which may require X amount of design effort. Each tile instance only requires X/4 amount of effort, but each instance can be used in $4^3=64$ different design instances. Therefore, the effective effort required to generate each tile instance is $X/(4*64)$, and each instance of the total tiled design requires X/64 amount of effort.

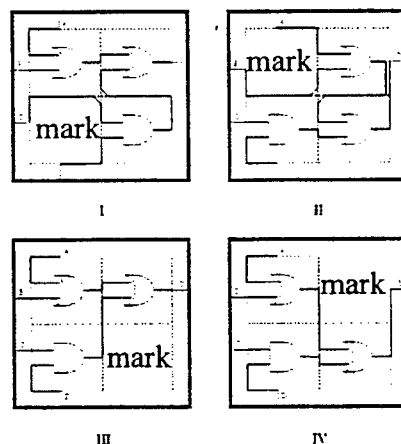


Figure 3. Four instances of the same function with fixed interfaces

1.3 Limitations

The standard digital design flow follows these steps: behavioral hardware description language (HDL), synthesis to register transfer language (RTL), technology mapping, and finally physical layout involving place and route. Marks can be applied to any level of this design flow and, if developed properly, will propagate to later stages. However, because a mark is nonfunctional, it may be removed by reverse engineering a design to a stage in the flow before the mark has been applied. Fortunately, most FPGA vendors will not reveal the specification of their configuration streams, specifically to complicate the task of reverse engineering and thus protect the investment of their customers. For example, the Xilinx XC4000

devices follow a form of Pareto's rule: the first 80% of the configuration information can be determined relatively easily by inspection, the next 16% is much more difficult, etc. The complexity is enhanced by an irregular pattern that is not consistent between rows or columns, as a result of the hierarchical interconnect network. Xilinx does not take any specific actions to make their configurations difficult to reverse engineer. However, they do believe that it is difficult to do in general, and they promise their customers that they will keep the bitstream specification confidential in order to raise the bar for reverse engineering [9].

1.4 Contributions

This paper presents the first fingerprinting method for protecting reusable digital circuit IP, even after the IP has been delivered in commercial products. By manipulating hardware resources, we are able to encode relatively long messages in a manner that is difficult to observe by a third party, resists tampering and collusion, and has little impact on circuit performance or size. This capability provides three main benefits:

1. It reduces the risk that a circuit will be stolen, i.e. used illegally without payment or transferred to a third party.
2. It identifies not only the origin of the design, but also the origin of the misappropriation.
3. It can be used to identify the backend tool chain used to develop a design, and thus be part of the royalty mechanism used for CAD tools.

1.5 Paper Organization

The remainder of the paper is organized in the following way. Section 2 discusses work related to this fingerprinting approach, and Section 3 details the approach itself. Section 4 evaluates the approach through experimentation, and the work is summarized in Section 5.

2 Related Work

Related work can be divided into three main areas: digital watermarking, fingerprinting, and reverse engineering.

Ad-hoc techniques for marking text and image documents have been manually practiced for many centuries [10]. Recently, techniques for hiding signature data in image, video, and audio signals have received a great deal of attention. A spectrum of steganographic techniques for protection of digital images has been proposed [11-13]. While many of the initial techniques for marking images were not able to provide proper proof of ownership [14], several recent techniques provide protection against all known attacks [15].

Although protection of digital audio signals emerged as a more difficult task, at least three different techniques have been proposed [12, 16, 17]. The protection of video streams using steganography has been demonstrated by several European and US research teams [18-20].

Recently, a set of techniques has been proposed for protection of digital circuit IP through watermarking at the behavioral level using superimposition of additional constraints in conjunction to those specified by the user [21]. In this paper, we

propose the first fingerprinting technique for FPGA designs. The majority of intellectual property is already programmable components, and all economic and historical trends indicate that the importance of these components will continue to rise. Finally, note that addressing the design at a lower level of abstraction has one additional advantage. All designs are significantly larger at lower levels of abstraction, enabling marks that are more difficult to detect and remove.

There has been a number of fingerprinting efforts reported in data hiding and cryptographic literature [22-24]. A spectrum of protocols has been established which greatly enhance the protection of both buyers and merchants of digital artifacts. All of these techniques are targeting protection of still artifacts, such as image and audio streams. To the best of our knowledge, this work is the first effort that addresses IP protection using fingerprinting.

While Xilinx and other FPGA vendors make some efforts to complicate the task of reverse engineering, it certainly is possible to crack the configuration specification with a concerted effort. NeoCAD Inc. was able to accomplish this for the Xilinx XC4000 series devices through a directed investigation of the output produced by the Xilinx backend tools. Given this information, it should be relatively straightforward to produce a Xilinx netlist file and then use commercial tools to move back up the design flow. Another line of attack involves removing the packaging material and successive layers and using image processing inspection to produce a circuit representation of the CLB. This approach has been used to produce a complete layout of a 386 microprocessor in approximately 2 weeks [25].

As a consequence of the proven success of reverse engineering, we believe that hiding the mark is necessary but not sufficient. Any effective fingerprinting scheme should make the mark appear to be part of the functional digital circuit to whatever extent is possible.

3 Approach

The watermarking technique is the general approach of inserting marks in unused CLB LUTs as introduced in the motivational example above. Results [5] indicate that the area and timing overhead required for inserting large marks in a design is low and that the approach is secure against most attacks.

Directly applying the watermarking technique to fingerprinting (i.e. replace the design origin signature with the recipient's fingerprint for each copy) is susceptible to collusion. Performing a simple comparison between the two bitstreams would reveal that the only differences were due to the individual fingerprints. Removing the differences would yield a fully functional yet unmarked circuit.

This vulnerability can be avoided by taking advantage of the flexible nature of FPGAs to create differences among functional components of the designs. By moving the location of the fingerprint for each instance of the design (i.e. reserve different CLBs for the fingerprint), the functional components will also have a different layout. Therefore, all comparisons that are done yield functional differences, and any attempt to remove the differences would yield a useless circuit.

However, generating an entire layout for each instance of the design would require a trip through the place-and-route tools for the entire circuit. Tiling requires that only a small portion of the design be changed. The algorithm divides a design into a set of tiles that possess the same characteristics as the example in Figure 3.

That is, each tile has specific functionality and a locked interface to the rest of the design. Several instances of each tile can be generated, and each instance can replace another without affecting the rest of the circuit (except timing) due to the locked interface. The various tile instances can then be matched to create one instance of the entire design. This reduces the total number of instances that need to be generated, and vastly reduces the effort and memory required to produce each instance.

The design tiling algorithm was originally developed in an eye toward fault-tolerance, but with the same goal of effort and memory reduction [6]. For fault-tolerance, different instances of each tile reserve different CLBs as unused. In the face of a CLB fault, the appropriate instance can be activated without affecting the rest of the circuit. The same result could be achieved by storing a great many instances of the entire design, leaving various CLBs free in each instance, but the effort to place and route each instance and the memory required to store each instance makes this approach impractical.

Much in the same way that tiling for fault-tolerance reduces the effort required to generate the various fault-tolerant instances and the memory to store them, tiling also makes fingerprinting more efficient and practical.

3.1 Watermark Preparation, Embedding, and Validation

We use cryptography tools to generate a set of FPGA physical designs (configurations) which correspond to the signature of the author of the design. The application of cryptographic techniques ensures also cryptographically strong hiding and low correlation of the added features.

The first step of the signature preparation and embedding process is encoding of the authorship signature as a 7-bit ASCII string. The signature string is given to the fingerprinting system for embedding in the circuit and is later produced by the verification program. The string is first compressed using a cryptographic hash function. The output is processed using public-key encoding. Finally, to produce a message that corresponds to the initial signature, we use a stream cipher. Note, in such a way, we generate a signature of arbitrary length.

Specifically, we use for the signature preparation the cryptographic hash function MD5, the public-key cryptosystem RSA, and the stream cipher RC4 [26, 27], on which many of today's state-of-the-art cryptographic commercial programs are based.

The next step in signature preparation involves adding error-correction coding (ECC). By introducing ECC into the signature, we combat the malicious third party that manages to identify a part of a signature and attempts to modify or remove it. A fundamental tradeoff exists between the number of bits allocated to the signature and ECC, and the sum must not exceed an estimate of the available bits free for encoding. We propose using a fault-tolerant encoding scheme that is similar to that used for marking file system structures on disk drives: the system decides upon a level of ECC protection based on the available free space and writes these settings into the available space. Using this approach, each design can have an appropriate amount of ECC while still guaranteeing that a generic verification system will be able to retrieve the signature.

The final step in signature preparation involves interleaving multiple ECC blocks. Consider the case where all signature information is encoded in the 16x1 LUTs of a Xilinx XC4000 device. It is possible that a malicious third party would be able to

identify a particular LUT that is non-essential to the device function and change its programming. If sixteen consecutive ECC blocks are interleaved, one bit at a time, over a set of LUTs, then each LUT will only contain one bit from any ECC. This interleaving guarantees that the validation software can successfully retrieve the signature in the face of any single point fault, i.e. a LUT that has been tampered with.

Validation involves retrieving the signature embedded in the configuration. The essential element to validating any signature is retrieving the FPGA configuration. This step is straightforward for FPGAs based on static memory, as they are loaded over an external bus that can be monitored. It may not be possible to retrieve the configuration for technologies based on anti-fuse or flash-memory, though the issue remains open given recent successful efforts at reverse engineering. For this reason, we have focused our technique on static memory devices.

When the owners of an IP block believes their property has been misappropriated, they must deliver the configuration in question to an unbiased validation team. The IP vendor produces a seed that they claim was used to produce the block. With the seed and signature, the validation team reverses the signature preparation and embedding process: identify the CLBs used for hiding the signature based on the specific tile instances of the suspected instance of the complete design, reverse the block interleaving, apply the ECC if necessary, decrypt the message using a known key, and finally print out the resulting signature. If the signature matches that claimed by the IP vendor, then the source of the misappropriation has been established.

Essentially, the owners must demonstrate that encoded constraints match their encrypted signature. Therefore, they must provide the validation team with their original and encrypted signatures. The encrypted signature can be verified to match the constraints in the realized design, but this procedure assumes that the third party (validation team) will not reveal the signature to others later. We plan to address zero-knowledge proofs for signatures (where this restriction is removed) in future efforts.

3.2 Fingerprinting

After each instance for each tile is generated, the instances are prepared for marking. Every unused CLB is incorporated into the design with unused network connections and neighboring CLB inputs, and timing statistics are generated for each instance. Depending on the timing specifications of the design, some instances may be discarded. The remaining instances are collected in a database. For example, MCNC benchmark c499 can be divided into 6 tiles, each with 8 instances, creating the possibility for $8^6 = 262,144$ different instances of the total design.

When a copy of the design is needed, an instance for each tile is extracted from the database and the recipient's fingerprint is inserted in the unused CLBs.

A group of people colluding may be able to find that they have instance matches among some of their tiles, thus allowing for tile comparison collusion, but it is extremely unlikely that matches will be found among all (or even a large portion) of the tiles. Furthermore, the tile structure and boundaries are not generally apparent to the colluders, as they are not an inherent property of the FPGA configuration. Therefore, the colluding recipients may be able to remove a portion of their fingerprints, but the majority of the fingerprints will remain intact. The key to this approach is efficiently introducing wide variation among the functional parts of the

designs, so that collusion cannot be used to separate common functional components from unique fingerprints.

The pseudo-code in Figure 4 summarizes the approach.

```

1.  create initial non-fingerprinted design;
2.  extract timing and area information;
3.  while (!complete) {
4.      partition design into tiles;
5.      if (!(mark size && collusion protection)) break;
6.      for (i=1; i<=# of tiles; i++) {
7.          for (j=1; j<=# of tile instances; j++) {
8.              create tile instance(i,j);
9.              if (instance meets timing criteria) {
10.                  incorporate unused CLBs into design;
11.                  store instance;
12.              } } } }
13.  for (i=1; i<=# of recipients; i++) {
14.      prepare mark(i);
15.      select tile instances from database;
16.      insert mark in unused LUTs;
17.  }

```

Figure 4. Pseudo-code for fingerprinting approach

Lines 1 and 2 initialize the process by establishing the physical layout for the non-fingerprinted design, on which all area and timing overhead is based. Lines 3-12 perform the tiling technique, creating a database of tile instances. The variables for this section are mark size, collusion protection (level of security based on presumed number of collaborators), and timing requirements. Mark size and collusion protection affect the tiling approach, while the timing requirements define the instance yield (i.e. individual tile instances are accepted contingent upon their meeting the timing requirements). Lines 13-17 are executed for each distributed instance of the design. Line 14 derives the unique recipient fingerprint with asymmetric fingerprinting techniques [23, 24].

3.3 Tiling Optimization

The tiling technique performed in lines 3-12 involves selecting a tile size (x) that best balances the security of the fingerprint and the design effort required for the fingerprinting process. The selection of x is based on a set of given constants and user defined variables, including the desired emphasis on either security or effort.

The constants are the FPGA device size (d) and the CLB utilization ($a = \# \text{ unused CLBs} / \# \text{ total CLBs}$). The variables defined by the user are the timing specifications (which impact instance yield (y), the percent of tile instances that meet the timing specifications), a collusion set size (n), and the maximum percentage of the mark that can be removed while leaving enough of the fingerprint intact for unique recipient identification (b). These five criteria lead to preliminary calculations: the number of tiles in the design ($t = d/x$), the number of free CLBs per tile ($f = \lfloor x * a \rfloor \approx x * a$), the

number of instances per tile ($i = \binom{x}{f}$), the number of instances per tile that meet the

timing constraints ($j = i * y$), and the design effort required for the fingerprinting technique ($e = t * x * i = d * i$). Design effort refers to the number of complete passes through the CAD tools (i.e. configurations of the entire design) that are required.

The above information can be used to calculate the odds that n people could collude to remove their fingerprints in one tile:

$$c_1 = 1 - \frac{(j)!}{((j-n)! * (j)^n)}$$

Equation 1. Odds to remove fingerprint from one tile

Equation 1 assumes that the only way to remove a fingerprint from a tile is to find two matching tile instances, thus making the recipient fingerprint the only difference between two tiles. A simple XOR comparison between the two tiles would remove the two fingerprints, thus creating an instance which can be distributed to all participating colluders. The owner's signature remains in all tiles, however.

The process of removing the fingerprint from one tile can be repeated until a certain portion of the complete design has been cleaned. A binomial calculation indicates the odds that $b\%$ of the fingerprint can be removed:

$$c_{\%} = \sum_{k=b*t}^t \binom{t}{k} * (c_1)^k * (1-c_1)^{t-k}$$

Equation 2. Chance to remove fingerprint from $b\%$ of the tiles

The security of the fingerprint improves with a larger tile size (x) as a result of the greater number of instances ($\binom{d}{f*t} > \binom{x}{f} * t$). However, as mentioned above, larger tile sizes require more effort (e) to generate. The product of security and design effort yields:

$$c_{\%} * e = d * \binom{x}{x*a} * \sum_{k=b*d/x}^{d/x} \left[\binom{d/x}{k} * \left(1 - \frac{[\binom{x}{x*a} * y]!}{[\binom{x}{x*a} * y - n]! * [\binom{x}{x*a} * y]^n} \right)^k * \left(\frac{[\binom{x}{x*a} * y]!}{[\binom{x}{x*a} * y - n]! * [\binom{x}{x*a} * y]^n} \right)^{(d/x) - k} \right]$$

Equation 3. Design effort times odds to remove fingerprint from $b\%$ of the tiles

This equation reveals that there is a critical value for x which balances the tradeoff between security and design effort. Certain applications may put a stronger emphasis on one or the other. For today's FPGA applications, design effort is one of the prime limiting factors. FPGA mapping is extremely time consuming, thus forcing an emphasis on limiting design effort. Also, today's applications see configuration bitstreams being distributed directly to approved recipients, thus requiring any collusion effort to bring n number of people together. Any one person would have a difficult time collecting a larger number of distributed instances of the design.

Conversely, trends in FPGA technology and applications may put a reduced emphasis on design effort and a greater emphasis on security. FPGA CAD tools continue to improve and workstations on which the software runs are improving tremendously. Therefore, in the future, design effort won't be as onerous. Also, one foreseeable application of FPGA designs includes distributing designs over the internet, either directly to recipients or available as a form of hardware application. In either situation, one person could gain access to a great many instances of the design, either by eavesdropping on a network line or downloading numerous instances of the application, each time receiving a different fingerprint. Then n becomes not the number of people colluding but the number of instances that are being compared, possibly by one person. This possibility raises the importance of fingerprint security.

4 Experimental Results

To evaluate the area and timing overhead of the approach, we conducted an experiment on nine MCNC designs. For design effort and fingerprint security, the following constants were assumed: device size (d) = 400 CLBs, number of unused CLBs/number of total CLBs (a) = 0.1, and instance yield (y) = 0.9.

The overhead of the proposed approach comes in the form of area (physical resources), timing, and design effort. Area overhead is inevitable, as previously unused LUTs are used to encode the mark. Table 1 shows the area of the designs before and after the application of the fingerprinting approach. A number of factors complicate the task of calculating the physical resource overhead. The place-and-route tools will indicate the number of CLBs that are used for a particular placement. However, these utilized CLBs rarely are packed into a minimal area. Unused CLBs introduce flexibility into the place-and-route step that may be essential for completion or good performance. For example, the initial c880 design possesses a concave region that contains 42 utilized CLBs but also 10 unutilized CLBs (19%). Therefore, we will report overhead in terms of the area used by the fingerprinted design minus the total area of the original design, including unused CLBs such as the 19% measure above. The average, median and worst-case area overheads were 5.4%, 5.3%, and 9.8% respectively. The size of the mark (signature + fingerprint) that can be encoded is dependent on this overhead. If a larger mark is desired, extra CLBs can be added thus increasing overhead.

Design	Original # of CLBs	Final # of CLBs	Final – Original Original
9sym	46	49	.065
c499	94	96	.021
c880	110	115	.045
duke2	93	100	.075
rd84	27	28	.037
planet1	95	100	.053
styr	78	81	.038
s9234	195	206	.056
sand	82	90	.098

Table 1. Variation of resources used among instances for each tile

Timing overhead may arise due to the constraints on physical component placement as defined by the size and location of the mark. A LUT dedicated to the mark may impede placement of circuit components and lengthen the critical path. As the mark size grows relative to the design size, more constraints are made on the placement of the design, thus increasing the possibility for performance degradation.

Timing overhead is show in Figure 5. For each design, the instance yield (i.e. number of tile instances that meet the timing specifications / total number of tile instances) is shown as the timing specifications (measured as percent increase over the original, non-fingerprinted design timing) grow more lenient. The results reveal that a 20% increase in timing yields approximately 90% of total tile instances as acceptable. Relatively small changes in a circuit netlist or routing constraints can often result in a dramatically different placement and a corresponding change in speed. It appears that the impact of fingerprinting on performance is below this characteristic variance.

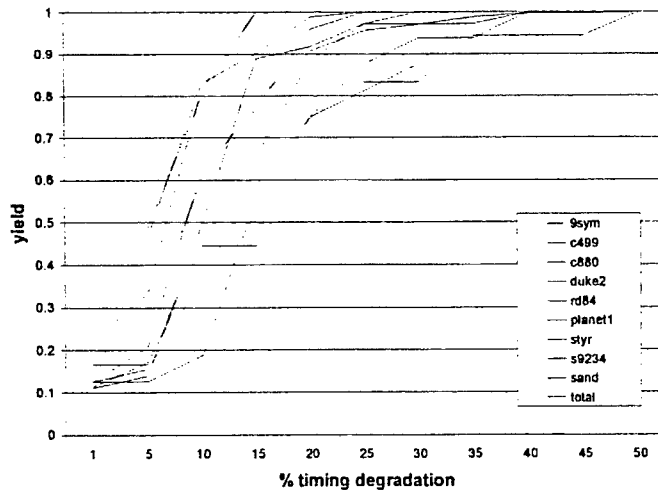


Figure 5 Instance yield vs. timing specifications

As the equations in Section 3.3 indicate, effort (e) increases with tile size (x). Design effort may often be the limiting factor in tile size selection, despite the fact

that security increases with tile size. By examining the chance that an increasing number of colluders have to break one tile (Equation 1) and all tiles (Equation 2 with $b=100\%$) by direct comparison collusion with varied tile sizes, it is clear that larger tile sizes drastically reduce the chance that any portion of the fingerprint may be removed. Even for a tile size of 40 and 200 people colluding, there is only one chance in approximately 5 million that the entire fingerprint could be removed. It therefore is important to compare the tradeoff between design effort and fingerprint security in a proper manner. Figure 6 is a direct multiplication comparison (Equation 3). Other comparisons may be more relevant based on the application and design setting. As mentioned above, current applications do not require as much security, as a large number of design instances aren't easily available to a group of colluders. Also, current design settings place a strong emphasis on design effort, as modern FPGA mapping technology is time consuming. Therefore, a different comparison (e.g. security * effort²) may present a more useful measure of current needs. As mentioned above, future applications may require a different comparison, perhaps placing a greater emphasis on fingerprint security and a smaller emphasis on effort. Figure 6, however, reveals that for a large number of colluders and direct multiplication comparison, it is better to have a larger tile size.

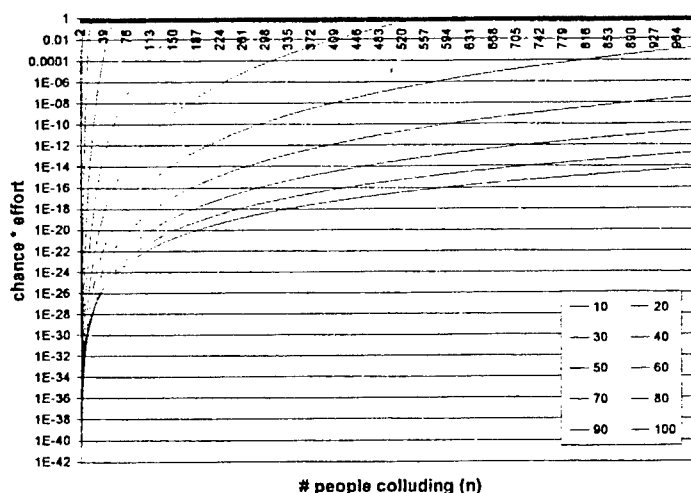


Figure 6. Chance to break all tiles * design effort

Figure 7 shows the same data as Figure 6, but it instead plots various collusion sizes against tile size and focuses in on a smaller, more reasonable number of colluders. The minimum value for each plot is the critical value denoting the optimal tile size for the particular collusion group size. It is easy to see here, that the best tile size actually is a mid-sized tile for a small collusion group (e.g. $n=2 \Rightarrow$ best $x=40$; $n=10 \Rightarrow$ best $x=80$), and the specific optimum tile size increases for larger collusion groups.

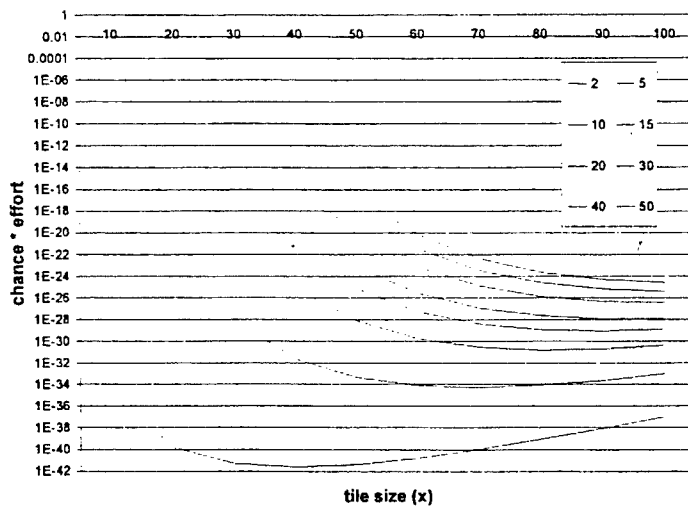


Figure 7. Tile size critical value

Figure 8 displays the chance that a growing number of colluders have to remove a certain percentage of the fingerprint for a tile size (x) of 40. Even for a small tile size such as 40, it remains extremely unlikely that a colluding group could remove even a small portion of the fingerprint. The chance that 15 colluders would be able to remove 30% of the fingerprint by comparison collusion is one chance in approximately 4 million.

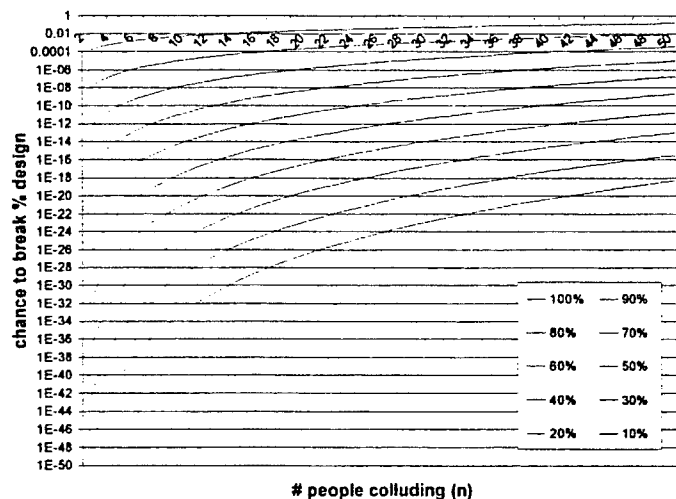


Figure 8. Chance to break % of fingerprint (tile size = 40)

Figure 9 directly multiplies security by design effort for each tile size revealing that the optimal tile size for a growing number of colluders is predominantly mid-sized tiles because the fingerprint security remains extremely high for mid-sized tiles while requiring significantly less design effort.

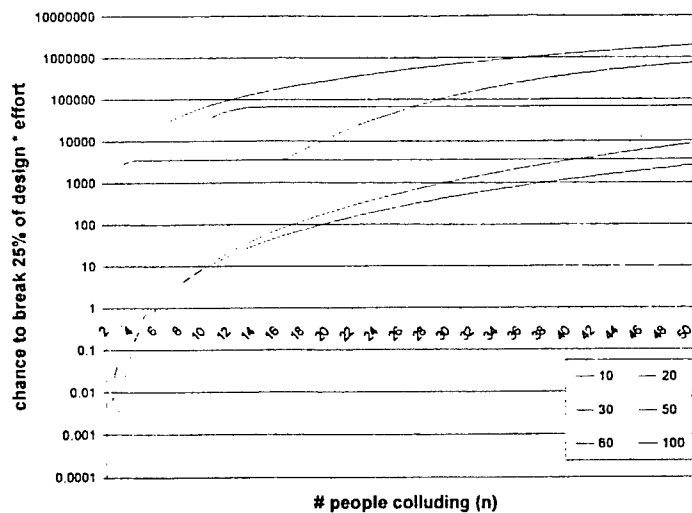


Figure 9. Chance to break 25% of fingerprint * effort

5 Conclusion

As digital IC design complexity increases, forcing an increase in design reuse and third party macro distribution, intellectual property protection will become more important. The fingerprinting approach presented here creates such protection for FPGA intellectual property by inserting a unique marker identifying both the origin and recipient of a design. The fingerprinting process produces an extremely secure mark (chance of removing a fingerprint is always less than one in a million) but requires little extra design effort. Although the mark is applied to the physical layout of the design by imposing constraints on the backend CAD tools, experiments reveal that the area and timing overhead is extremely low.

Acknowledgements

The authors would like to thank Gang Qu for his assistance. This work was supported by the Defense Advanced Research Projects Agency of the United States of America, under contract DAB763-95-C-0102 and subcontract QS5200 from Sanders, a Lockheed Martin company.

References

- [1] J. Turley, "ARM Grabs Embedded Speed Lead," *Microprocessor Report*, vol. 10, 1996.
- [2] J. Montanaro et al., "A 160MHz 32b 0.5W CMOS RISC Microprocessor," *Proc. of International Solid-State Circuits Conference*, 1996.
- [3] S. Furber, *ARM System Architecture*, Menlo Park: Addison-Wesley, 1996, p. 329.
- [4] I. Ayres and S. D. Levitt, "Measuring Positive Externalities from Unobservable Victim Precaution: An Empirical Analysis of Lojack," *The Economics Review*, 1997.

- [5] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "Signature Hiding Techniques for FPGA Intellectual Property Protection," submitted to *ICCAD '98*, 1998.
- [6] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "Low Overhead Fault-Tolerant FPGA Systems," *IEEE Transactions on VLSI*, vol. 6, 1998.
- [7] Xilinx, *The Programmable Logic Data Book*, San Jose, CA, 1996.
- [8] Programmable Electronic Performance Group, "PREP Benchmark Suite #1, Version 1.3," Los Altos, CA, 1994.
- [9] S. Trimberger, Personal Communication, Xilinx Corporation, 1997.
- [10] H. Berghel and L. O'Gorman, "Protecting Ownership Rights Through Digital Watermarking," *IEEE Computer*, 1996, pp. 101-103.
- [11] J. Brassil and L. O'Gorman, "Watermarking Document Images with Bounding Box Expansion," *First International Workshop on Information Hiding*, Cambridge, U.K., 1996.
- [12] I. J. Cox et al., "Secure Spread Spectrum Watermarking for Images, Audio and Video," *International Conference on Image Processing*, 1996.
- [13] J. Smith and B. Comiskey, "Modulation and Information Hiding in Images," *First International Workshop on Information Hiding*, Cambridge, U.K., 1996.
- [14] S. Craver et al., "Can Invisible Watermarks Resolve Rightful Ownership?" *The International Society for Optical Engineering*, 1997.
- [15] A. H. Tewfik and M. Swanson, "Data Hiding for Multimedia Personalization, Interaction, and Protection," *IEEE Signal Processing Magazine*, 1997, pp. 41-44.
- [16] W. Bender et al., "Techniques for Data Hiding," *IBM Systems Journal*, vol. 35, 1996, pp. 313-336.
- [17] L. Boney et al., "Digital Watermarks for Audio Signals," *International Conference on Multimedia Computing and Systems*, 1996.
- [18] G. A. Spanos and T. B. Maples, "Performance Study of a Selective Encryption Scheme for the Security of Networked, Real-Time Video," *International Conference on Computer Communications and Networks*, 1995.
- [19] F. Hartung and B. Girod, "Copyright Protection in Video Delivery Networks by Watermarking of Pre-Compressed Video," *ECMAST '97*, 1997.
- [20] F. Hartung and F. Girod, "Watermarking of MPEG-2 Encoded Video Without Decoding and Re-Encoding," *Multimedia Computing and Networking*, 1997.
- [21] I. Hong and M. Potkonjak, "Behavioral Synthesis Techniques for Intellectual Property Protection," unpublished manuscript, 1997.
- [22] D. Boneh and J. Shaw, "Collusion-Secure Fingerprinting for Digital Data," *CRYPTO '95*, 1995.
- [23] I. Biehl and B. Meyer, "Protocols for Collusion-Secure Asymmetric Fingerprinting," *STACS '97, 14th Annual Symposium on Theoretical Aspects of Computer Science*, 1997.
- [24] B. Pfitzmann and M. Waidner, "Anonymous Fingerprinting," *International Conference on the Theory and Application of Cryptographic Techniques*, 1997.
- [25] R. Anderson and M. Kuhn, "Tamper Resistance - A Cautionary Note," *USENIX Electronic Commerce Workshop*, 1996.
- [26] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. New York: John Wiley & Sons, 1996.
- [27] A. J. Menezes, P. C. V. Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.

REAL-TIME IMAGE FORMATION EFFORT USING QUADTREE BACKPROJECTION AND RECONFIGURABLE PROCESSING

Matthew C. Cobb
James McClellan
Seung-mok Oh

Center for Signal and Image Processing
Georgia Institute of Technology
Atlanta, GA 30332

Mark Falco

Sanders, A Lockheed Martin Company
65 River Road
Hudson, NH 03051

Jeff Sichina

U.S. Army Research Laboratory
2800 Powder Mill Road
Adelphi, MD 20783

ABSTRACT

A combined effort is underway to demonstrate real-time processing of Ultra Wideband, Ultra Wide Angle SAR. The Quadtree Backprojection algorithm will be implemented using recently developed reconfigurable processors. The Quadtree Backprojection algorithm's computational savings will combine with the reconfigurable processor's ability to exploit parallelism and overcome a bottleneck within the algorithm to produce a fast, low-artifact image formation solution.

INTRODUCTION

Low frequency, imaging radar offers the potential of breakthrough capabilities for the warfighter. The reliable detection of obscured targets such as tactical vehicles hidden by tree canopy and mines buried underground is becoming an evermore significant need on the modern battlefield.

ARL, working in collaboration with several defense organizations, has been evaluating the utility of an Ultra Wideband (UWB), Ultra Wide Angle radar to perform effectively in these mission areas (1,2). In this defense-wide effort, enormous progress has been made in developing wideband radar technology and in improving our understanding of the phenomenology of targets and clutter within this frequency regime. This has prompted the construction of an airborne UWB radar demonstrator (managed by CECOM for DARPA) to be used for foliage penetration. Yet challenges remain.

For the warfighter, low-metal content anti-tank and anti-personnel mines are amongst the most difficult of targets - limiting freedom of maneuver because of their inability to be found. To have any chance of detecting such small targets using radar, virtually every aspect of the system design must approach ideal. Perhaps no area is more critical than image formation. By reducing the radar resolution interval so that it is commensurate with (or

smaller than) the target size, contrast (against the clutter background) is gained. However, both the computational complexity as well as the possibility of artifacting grows with enhanced resolution.

In this paper we discuss two synergistic undertakings aimed at providing high quality image formation in an efficient mechanization. The two efforts are a novel decomposition of a back projector-based image reconstruction method (which is known to have extremely low artifact levels) and the application of an emerging technology - reconfigurable processing - to this imaging method.

QUADTREE BACKPROJECTION

Conventional Delay and Sum Backprojection involves summation along hyperbolic contours in the space-time domain. The following equation defines an image point using the conventional backprojection method:

$$S(x,z) = \sum_{\alpha=0}^{L-1} D_{\alpha}(t - T_{x,z,\alpha})$$

$D_{\alpha}(t)$: Data corresponding to element α

x, z : Spatial position

$T_{x,z,\alpha}$: Delay associated with position (x,z) , aperture α

Delay and sum backprojection offers advantages in simplifying motion compensation and localizing processing artifacts. However, it requires more computations (order N^3) than competing algorithms (order $N^2 \log N$). This disadvantage sometimes requires the use of faster algorithms that approximate the backprojection imager.

The Quadtree Backprojection algorithm was introduced in [1] as an $N^2 \log N$ solution with the same advantages as conventional delay and sum backprojection. In the radix-2 case¹ the algorithm at each iteration combines pairs of ap-

¹ A generalized multi-radix algorithm is introduced in [2]

ertures into virtual apertures by beamforming, while partitioning the image field into quadrants. For each iteration, the number of apertures is halved, the number of image partitions is quadrupled, and the number of samples per image partition per aperture is approximately halved. If the process is repeated to the last iteration, a single virtual aperture will hold image data for each pixel². See Figure 1.

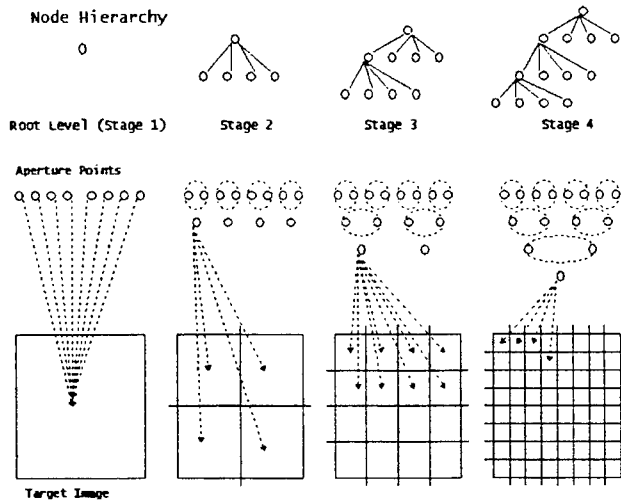


Figure 1. Quadtree Splitting Structure

SIMPLIFIED DISTANCE CALCULATIONS

The primary step in the quadtree algorithm, beamforming, is a multiply-add operation that is easily implemented in hardware. The beamforming cannot be accomplished, however, without an accurate distance measurement from each newly defined aperture point to each range cell in the newly formed image partitions. Calculating this 3-D distance efficiently becomes extremely important in building a fast imager, especially when the calculation is made without the benefit of floating point processors. Attention must be paid to scaling and word-length as well as computational cost. At Georgia Tech we have developed a preliminary solution for the distance calculations which successfully produce images using 15-bit distance measurements.

Each index calculation uses the previously saved distance between an aperture point and a ground patch center. (It is not necessary to save all distance measurements, just the distances to the ground patch centers.) This stored value,

² The data at any iteration can be handed-off to another algorithm (e.g. conventional backprojection or ω -k) to complete processing. This is generally advantageous after a few iterations.

along with the distance from each range-cell to the ground patch center can be used with the aperture and ground patch positions to calculate the necessary distances efficiently.

One of the methods that originated in [1] uses the Law of Cosine Formula. This method avoids a Square Root operation so that it is computationally efficient in a 2-D scenario. However, in recorded field data the aperture points vary in height, as well as in a 2-D plane, so the 3-D distance must be found. The simple method, which worked in the 2-D environment, is no longer valid. We present here an alternative calculation based on a vector analysis of the 3-D coordinates of the aperture points and the ground patch locations. This calculation exploits redundancy in calculating successive distances from a single aperture to multiple range bins in an image partition.

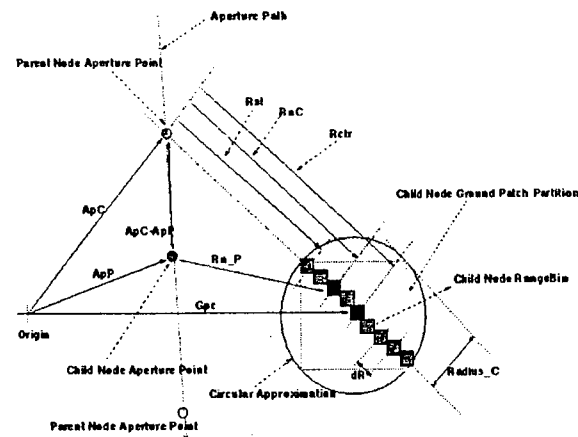


Figure 2. Distance Calculations

The following calculations assume that the child virtual apertures A_{PC} , at each stage are located exactly between the two parent apertures. Therefore it is only necessary to use one parent aperture A_{PP} .³

First calculate the range from each ground patch center to each parent aperture:

$$R_{ctr} = G_{PC} - A_{PP} \quad R_{st} = R_{ctr} - Radius_C$$

The desired result is $\|R_{st}\|$, the distance from an image point to the parent apertures:

³ This is not always true in the generalized quadtree algorithm. While the general case requires only slightly more computation, it introduces more notational complexity.

$$R_{np} = (A_{pc} - A_{pp}) + \alpha_n \times R_{ctr}$$

Where α_n is the ration:

$$\alpha_n = \frac{\|R_{nc}\|}{\|R_{ctr}\|} = \frac{\|R_{st} + n \times \Delta R\|}{\|R_{ctr}\|}$$

Since ΔR and R_{st} are in the same direction:

$$\alpha_n = \frac{\|R_{st}\|}{\|R_{ctr}\|} + n \times \frac{\|\Delta R\|}{\|R_{ctr}\|}$$

Suggesting:

$$\alpha_0 = \frac{\|R_{st}\|}{\|R_{ctr}\|} \quad \Delta \alpha = \frac{\Delta R}{\|R_{ctr}\|}$$

$$\alpha_n = \alpha_0 + \Delta \alpha \times n$$

This indicates a redundancy that can be exploited by:

$$R_{nc} = \alpha_n \times R_{ctr} = (\alpha_0 + \Delta \alpha \times n) \times R_{ctr}$$

And since,

$$R_{np} = \alpha_n R_{ctr} + A_{pc} - A_{pp}$$

$$R_{ctr} = G_{pc} - A_{pp}$$

We have,

$$\begin{aligned} \|R_{np}\|^2 &= \|A_{pc} - A_{pp} + \alpha_n \times R_{ctr}\|^2 \\ &= (A_{pc} - A_{pp})^T (A_{pc} - A_{pp}) \\ &\quad + \alpha_n R_{ctr} (A_{pc} - A_{pp}) \\ &\quad + \alpha_n R_{ctr}^T (A_{pc} - A_{pp}) \\ &\quad + \alpha_n^2 R_{ctr}^T R_{ctr} \end{aligned}$$

Now define:

$$\begin{aligned} A &= (A_{pc} - A_{pp})^T (A_{pc} - A_{pp}) \\ B &= R_{ctr} (A_{pc} - A_{pp})^T + R_{ctr}^T (A_{pc} - A_{pp}) \\ C &= R_{ctr}^T R_{ctr} \end{aligned}$$

So that

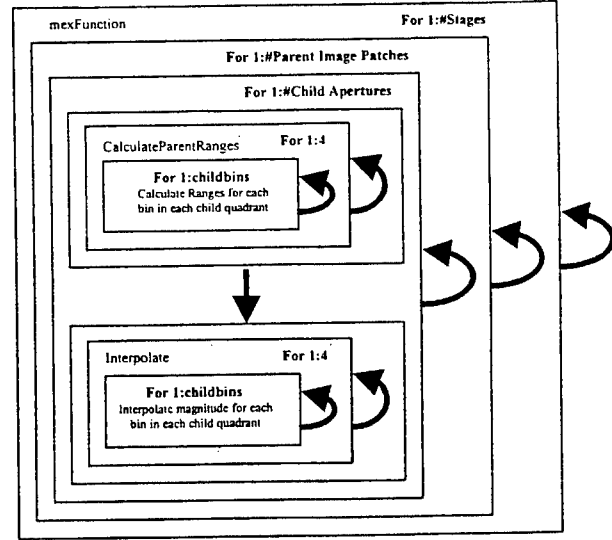


Figure 3: Algorithm Nesting Structure

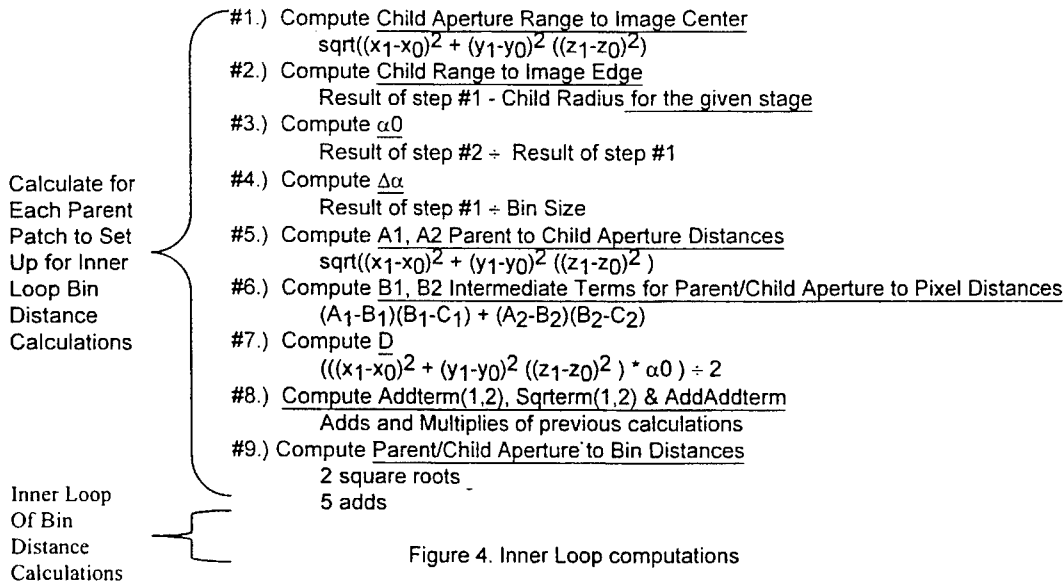
$$\|R_{np}\|^2 = A + \alpha_n B + \alpha_n^2 C$$

Note in this final equation that A, B, and C are constant for all the range bins in a give image partition for a single aperture. This allows for fast computation of $\|R_{np}\|^2$, since successive values can be computed with only two adds. The necessary square root calculation can be efficiently combined with the calculation of $\|R_{np}\|^2$ by using piecewise polynomial approximations.

HARDWARE IMPLEMENTATION

One of the challenges for ARL is in implementing algorithms efficiently on desired surveillance or weapons platforms. For example, a tactical UAV may have a few hundred watts of power, a cubic foot of volume and a 50-75 pound payload restriction. Therefore, one of the goals for this work is to explore implementations that combine algorithms and architectures to address these stringent constraints. This section describes our approach to mapping the Quadtree algorithm onto an architecture that is driven by the application constraints.

The control flow of the quadtree algorithm is organized as a sequence of loops, with the number of iterations per loop changing at each stage of the recursion. Fig. 3 shows the nesting structure of the algorithm. Furthermore, distance calculations are needed for the indices that point to the



data to be interpolated. Fig. 4 gives the form of the calculations for the inner loops of Fig. 3.

With this algorithm structure one can partition the computation by apertures pairs. For example, with 2^M apertures and 2^N processors ($M \gg N$) you can give each processor 2^{M-N} apertures pairs to work on recursively without any data sharing between nodes. This would result in 2^N child apertures, which could be processed with 2^L processors ($N > L$), again resulting in a partition of 2^{N-L} aperture pairs per processor and 2^L child apertures. It is interesting to note that as the recursion continues, the number of apertures halves and the number of patches quadruples. The total amount of data remains about the same.

Since the ground patch size decreases at each iteration, each processor winds up doing more work in the index calculations and less interpolation. This has been observed by the algorithm developers during simulation, one processor a bottleneck occurs. The first reaction is to find some way to adaptively partition the work by image patches. The problem here is that each image patch requires access to every aperture pair. Thus the memory or memory bandwidth requirement to be able to store or move all the apertures across the processing nodes would explode.

Our approach to accelerating the Quadtree is to optimize the calculations of the indices up to a particular stage in

the algorithm and then to switch to another algorithm like ω -k or backprojection which would again be accelerated on a single node. For example, the ω -k algorithm is FFT based. We may be able to do enough iterations of the Quadtree so that the FFT's are small enough to be run on individual processors (realizing that some data movement between processors will be necessary).

CONTEXT SWITCHING RECONFIGURABLE PROCESSING

To implement the acceleration we need something that can change rapidly at the operation as well as algorithm level. Typical implementations use commercial off the shelf (COTS) multi-processing systems. However, the generalized architecture of COTS equipment introduces a computational overhead which may result in more resources needed to solve the problem, thus making the overall solution unoptimized in size, power or weight. Sometimes a custom pre-processor or accelerator is used in conjunction with COTS to overcome this situation. This is where our research falls.

Field Programmable Gate Arrays (FPGA's) have been used for a number of years, in a number of architectures to accelerate applications. The acceleration is due to the fact that the FPGA can be customized to implement all or some of the application in hardware. The best acceleration can be achieved with custom hardware, but the advantage of using FPGA's is that the customization is re-

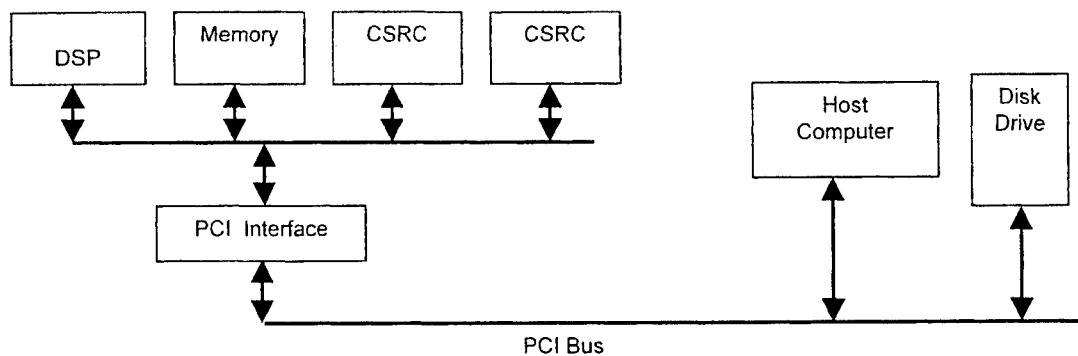


Figure 5: Demonstration System Block Diagram

programmable. This means that an architecture may be used to accelerate several different applications. Current FPGA solutions do best with pipelined data flow applications and act as either a pre-processor or as a co-processor customized for the particular application.

Because of the complex control flow, the Quadtree Algorithm does not lend itself well to being accelerated with current FPGA technology. However, it appears that the Quadtree algorithm would be a good candidate for Context Switching Reconfigurable Computing (CSRC) technology being developed by Sanders under the DARPA Adaptive Computing Systems program.

CSRC is a new direction in FPGA technology. Think of CSRC as having multiple-layers of logic, called contexts, which you can switch between on a single clock cycle. Each context implements a particular function (Arithmetic, DSP, I/O, etc.). As the algorithm executes, it switches context to accelerate a given function. In this manner one can use the CSRC device to rapidly switch functions at the operation level. Note that operations may be as simple as multiple adds or as complex as an entire DSP function. The device allows for data to be shared between contexts and also has RAM to implement local data storage.

As an example of how the CSRC might work, consider a two CSRC architecture being controlled by a general purpose processor (GPP). The GPP would manage the stage of recursion, the number of iterations and the interpolation. The GPP would load the CSRCs with desired parameters to compute the indices and retrieve the indices when calculated. A data dependency diagram is used to map the index calculations into the contexts of each of the CSRCs. As the quadtree algorithm details evolve, a dependency

diagram is updated in an effort to balance processing between two CSRCs, keeping each busy.

For further details related to CSRC technology, reference [3]. To drive our architectural design we will use the imaging geometry and data rates for a counter mine tactical UAV application. We will define real time as the ability to image one ground patch for every synthetic aperture flown (with latency). Our goal is to demonstrate that the CSRC can accelerate the Quadtree algorithm by rapidly changing at the operation level and then that it can be used to adaptively switch to another algorithm to accelerate the overall application.

Figure 5 is a high-level block diagram for the demonstration system. Currently, we are in process of developing the architecture for the Reconfigurable Computing Module (RCM) that includes 2 CSRCs, memory, DSP and a PCI interface. We are looking into developing the RCM as a PCI Mezzanine Card (PMC) that can be mounted on COTS carrier cards in either VME, Compact PCI or PC form factors.

We are developing a single node to benchmark the acceleration of the index calculations for the inner loops. Research will continue to find the optimum switching point and algorithm to run to overcome the limitations of the quadtree algorithm. We may then look at adaptively changing algorithms to complete the processing. At that point we could extrapolate to a system that could implement the entire application to understand the implications on size, weight and power for the target tactical UAV system.

REFERENCES

1. Martin Rofheart and John McCorkle, "An order of N^2 * Log(N) Back Projection Algorithm for Focusing Wide Angle Wide Bandwidth Arbitrary-Motion Synthetic Aperture Radar"
2. Seung-mok Oh, James McClellan, "Multi-Resolution Mixed-Radix Quadtree SAR Image Focusing Algorithms" 3rd Annual Fed Labs Symposium, Feb. 99.
3. "The Design and Implementation of a Context Switching FPGA" p. 78 of the IEEE Symposium proceedings on FPGAs for Custom Computing Machines, April 15-17.

*The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon

PROTECTING OWNERSHIP RIGHTS OF A LOSSLESS IMAGE CODER THROUGH HIERARCHICAL WATERMARKING

Hea Joung Kim
Electrical Eng.
U. of California
Los Angeles, CA

William H. Mangione-Smith
Electrical Eng.
U. of California
Los Angeles, CA

Miodrag Potkonjak
Computer Sci.
U. of California
Los Angeles, CA

Abstract - Current market forces make it necessary for designers to protect their work against illicit use. Digital watermarks can be used to sign a design and thus establish ownership. We present a hierarchical set of techniques for intellectual property protection of a linear predictive image coder. Watermarking techniques employed include switching entries in the Huffman coding table, applying zero cost hardware transformations, embedding a signature in the scheduling of shared hardware resources, and applying a watermark to the physical layout. Using these methods, it is possible to watermark complete ASIC or FPGA designs with little overhead in performance or achieved compression rates.

1 INTRODUCTION

Recently, there has been an increased interest in the area of intellectual property (IP) protection due to threat of piracy and counterfeiting. We have developed a hierarchical approach that can be used to watermark a complex block of IP at multiple levels of the design process. These sorts of techniques are evaluated in the context of a lossless image coder [1]. By watermarking the image coding hardware a designer will be able to assert ownership rights in the face of theft, even if sophisticated means are employed. The next few paragraphs will introduce the steps involved (Figure 1).

The assumptions made for the following hardware design is that the images are 256 by 256 arrays of 8-bit gray scale pixels. Furthermore, the training set used to construct the Huffman table for the errors of the linear prediction are F15, Bob, Lena, Football1, Football2, and Trit (Appendix A).

The linear predictive model employed here uses the pixel to the left, diagonal to left and above, above, and diagonal above to the right (Figure 1). This approach facilitates data re-use during raster scan processing. As shown in Figure 1, the first step in constructing a coder is to obtain the $error(x,y)$ distribution of the training set using the $model(x,y)$ and the $image(x,y)$. A linear prediction code is used for the entire image except the first column, first row, and last column of the image, which are predicted using the previous row, column, and row, respectively. The linear prediction can easily be modified for the predictors used in JPEG [2] by performing similar operations on a different set of pixels as described in the JPEG standard.

	x →		
y	A	C	E
	B	D	

Figure 1. Pixels used for linear prediction error $(x,y) = \text{Image}(x,y) - \text{Model}(x,y) = D - \frac{1}{4}*(A+B+C+E)$

An identified set of representative training images is used to generate a probability density function (PDF) for the error values ranging from -128 to 127. Each error value, ranging from -128 to 127, has a probability of occurrence with the 0 error (i.e. the model is correct) having the highest probability. The probabilities are used to generate the Huffman tree as shown in Figure 2. The first step is to get the probabilities of all the error symbols. The second step is to combine symbols having lowest probability into a new symbol node with the original nodes as children. This step repeats until a single tree remains [3].

The Huffman tree encodes each error values to some specific number of code bits. This coding scheme allows low probability errors to be coded with a large number of bits while those with high probability (e.g. '0') get coded with a fewer number of bits. The upper-left pixel and a sequence of variable length error values represent coded images.

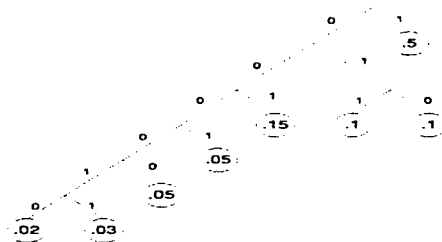


Figure 2. Huffman Tree Example

Watermarks are applied by modifying the table used to construct the Huffman tree, the labels applied to edges, the hardware for computing the model value, the order of computing intermediate terms, and finally the finite state machine (FSM)

1. Get probability distribution function of linear prediction errors for specified training set of images
2. Watermark the symbol (error) table by swapping specific entries or extending the code length and the code itself with edge labeling (depth first search)
3. Watermark hardware using functionally equivalent logic structure
4. Watermark controller using scheduling of shared hardware resources
5. Watermark state machine
6. Watermark physical FPGA design

Figure 3. Design Hierarchies Supporting Watermarks

used for control. Each technique will be discussed in the following sections.

2 RELATED WORK

With the wide spread use of the internet, many valuable digitized images require some form of title certification. Quite a few techniques have been developed for watermarking images [4], video [5], and audio [6]. Unlike this work, these techniques involve watermarking the media, and not the media codec.

There have been numerous papers on linear and non-linear prediction for lossless image compression [7,8] from different application domains. Most of these papers describe the different x,y coordinate pixels to use to do linear prediction more efficiently for the images of different types, e.g. medical images, synthetic aperture radar images, infrared images, and others. All these papers present a method to predict the errors from a smaller range so the prediction is easier. The techniques presented here are directly applicable to this broader domain.

Recent research results have investigated a range of hardware watermarking techniques [9,10] at a number of distinct levels. The work present here is the first multi-level system, and the first attempt to watermark an application algorithm (i.e. linear prediction image compression).

3 APPROACH

The following sections will describe all the steps involved in a complete top-down watermarking solution.

3.1 Watermarking the Compression Code

The first step is to create a look-up table for the Huffman coding of the error(x,y) distribution. The Huffman table is generated by the summed errors of the training set images. The probability density function is calculated to generate the Huffman table of the image collection. The PDF of each error (ranging from -128 to 127 given an 8-bit gray scale image) is found using a C program. Following the algorithm for constructing the Huffman code, the pair of the least probable errors is iteratively combined to generate the Huffman tree.

An arbitrary sequence of length 36 and 72 signature can be embedded into the Huffman table. The simple approach is to take the first 8-bits of the signature to select one of the probabilities of the error symbols listed in decreasing order. Then if the ninth bit is a 1, the probabilities of the error below and above the 8-bit number are swapped. If the ninth bit is a 0, the probabilities of the selected entry and the immediately preceding locations are switched. If the code length is the same for the two probabilities that will be switched, the error code is extended by one bit to embed a '0' or '1'. Thus, this approach may result in increasing the code length of certain error symbols. Figure 4 shows the method described.

1. Take the first eight bits to get the location of decreasing probabilities of the error symbols
2. Take the ninth bit to swap the probabilities and error code (if the error code is the same length swap and extend 1 bit)

3. 01110100,0 (116,0) the 116th and 117th are switched 116th probability code is 11 bits and is extended to 12 with the 12th bit being a '0'
4. 00000011,1 (3, 1) the 2nd and 4th are swapped and 3rd probability code is lengthened to 6bits with the 6th bit being a '1'
5. 10100000,1 (160,1)
6. 00000000,1 (neglect...too expensive to change location 0 or 1)
7. 11010000,0 (208,0)
8. 00000010,1 (2,1) ..neglect since duplicated
9. 00000111,0 (7,0)
10. 00011110,0 (30,0)

Figure 4. Watermarking the Compression Code

Loc	Size	error	trit	lena	fball2	fball1	F15	bob	prob
0	5	0	28795	27315	35140	34935	54045	101095	0.14308
1	5	255	9120	24160	14870	14710	10105	5270	0.03979
2	5	1	8910	23335	13535	14470	1510	8890	0.03593
3	5	254	8255	21580	13420	13440	2440	8270	0.03428
4	5	4	8425	12855	10280	10335	15845	8655	0.03377
5	5	5	7560	10120	9275	9075	24360	5580	0.03355
6	5	2	8525	19700	12590	12730	1905	6980	0.03175
7	5	250	8540	7910	8965	8630	19685	7460	0.03112
8	5	253	8585	17065	11875	12630	895	9915	0.03100
9	5	251	7530	9605	9985	10040	14565	6665	0.02969

Table 1. Original Huffman Table with no watermarking

Loc	Size	error	trit	lena	Fball2	fball1	F15	bob	prob
0	5	0	28795	27315	35140	34935	54045	101095	0.143
1	5	255	9120	24160	14870	14710	10105	5270	0.039
2	5	1	8910	23335	13535	14470	1510	8890	0.03593
3	6*	254	10692	28002	16242	17364	1812	10668	0.034
4	5	4	8425	12855	10280	10335	15845	8655	0.03377
5	5	5	7560	10120	9275	9075	24360	5580	0.033
6	5	2	8525	19700	12590	12730	1905	6980	0.03175
7	6*	250	10248	9492	10758	10356	23622	8952	0.031
8	5	253	8585	17065	11875	12630	895	9915	0.031
9	5	251	7530	9605	9985	10040	14565	6665	0.029

Table 2. Huffman Table with Watermarking

Table 1 and Table 2 show the exact procedures taken to embed a signature in the Huffman table for the two examples listed in Figure 4. The first example is where the first 8-bits are a 3 and the 9th bit is a '1', while the second example is where the first 8-bits are a 7 and the 9th bit is a '0'.

The assignment of the zero and one to the Huffman tree edges can be used to watermark the codes of the error symbols. We set a rule that two nodes emerging

from an interior node will be sorted so that the sub-tree with the lowest probability will be assign an edge value of zero. If the actual edge assignment follows this rule it encodes a '0', otherwise it encodes a '1'. Thus, if the training set is known, a signature can be retrieved through systematic probing.

3.2 Watermarking the Datapath

The hardware is watermarked at multiple stages of the design (Appendix B). The first step involves transformations as shown Figure 2. The model(x,y) is obtained by $\frac{1}{4} * (\text{Image}(x-1,y-1) + \text{Image}(x-1,y) + \text{Image}(x+1,y) + \text{Image}(x,y-1))$. The basic operations needed are to shift and add or to add and shift.

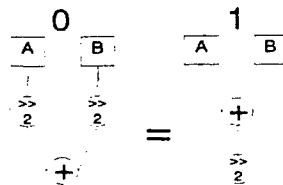


Figure 5. Transformations for embedding 0 or 1

Figure 5 shows that a shift-add operation designates a '0' and an add-shift operation designates a '1'. A parallel datapath can be used to embed a 10-bit signature as shown on Figure 6. The hardware markings discussed in sections 3.2, 3.3, and 3.4 are more difficult to extract after a design appears in the field. Unlike modifications to the coding structure, hardware marks cannot easily be extracted through external probing. Nonetheless, techniques do exist for extracting RTL and datapath information using optical inspection [11].

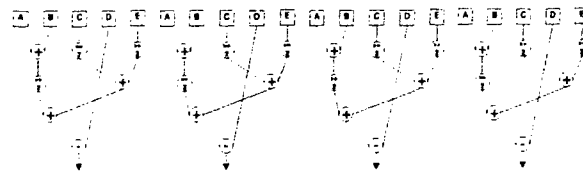


Figure 6. Signature '10101010'

3.3 Watermarking the Control Structure

The next form of signature embedding is to use scheduling to insert artificial dependencies between operations in the datapath (Figure 7). An operation delayed even when it could have been executed in parallel embeds a '1' (operations -gray filled - Figure 7), while an operation that follows the normal execution flow embeds a '0'.

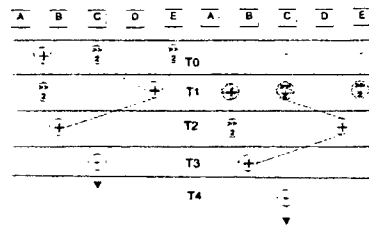


Figure 7. Scheduling to embed "000111" a signature

3.4 Watermarking the Finite State Machine

Following hardware transformations and datapath scheduling, another signature can be embedded using the states for the finite state machine that controls the flow graph of the operations shown above. The state numbers for the state machine can be created from an arbitrary sequence of numbers.

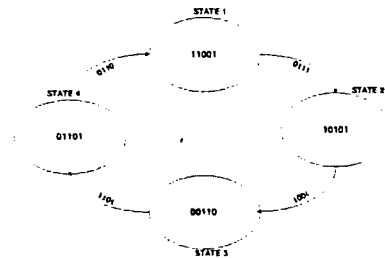


Figure 8. Finite State Machine used to embed signature

Figure 8 shows exactly how another signature can be embedded in the design. This is not a very rigorous or undetectable signature but it would be easy to demonstrate ownership if the design is copied with no thought. The signature would read '11001->10101->00110->01101'.

The entire design would be specified in Verilog and synthesized to an FPGA.

4 EXPERIMENTAL RESULTS

The experimental results available are for the Huffman table used for compression of the image data and the cost associated with embedding a signature. The raw performance rates are not interesting given that any hardware implementation will be faster than a software implementation. The Verilog code embeds signatures using scheduling, transformations, states in a FSM, and jump conditions signatures. Thus, the focus of the experimental results is on the cost of coding an image using the Huffman table with a signature embedded in the table.

The first set of data shows the best compression possible assuming that errors with value greater than 20 can be coded using 20 bits. Despite the fact that it is possible to code any error greater than 10 with a fixed bit-length of 9, the method of coding errors greater than 20 bits can be just as effective in showing the cost of embedding a signature in the Huffman table.

Figure 9 shows the different sizes of the compressed images for the different types of images. The best code represents a Huffman tree generated using each images probability density function to compress the same image. The summed Huffman is the code generated from the sum of all the PDFs of the images. The 36-bit signature is the size of the compressed image given that 36 bits of data has been embedded into the summed Huffman code using the method described in Figure 4. Figure 9 shows a graph comparing the results.

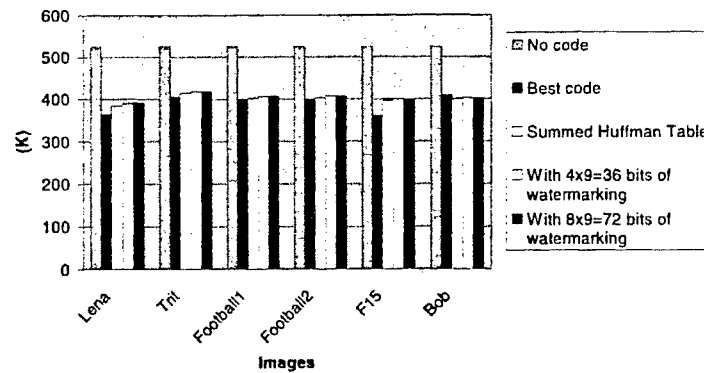


Figure 9. Comparison of image coders (uncompressed, compressed, and watermarked)

Figure 9 and Figure 10 show the percentages that the image sizes increases or decreases relative to the summed Huffman table, non-watermarked, and non-compressed images. It is important to note that the cost of watermarking is less than 2% of the compressed image size and in some cases even better since the summed Huffman table is not optimal for a specific image. In some instances, the size is smaller after the watermarking due to the switched probabilities from watermarking the code table.

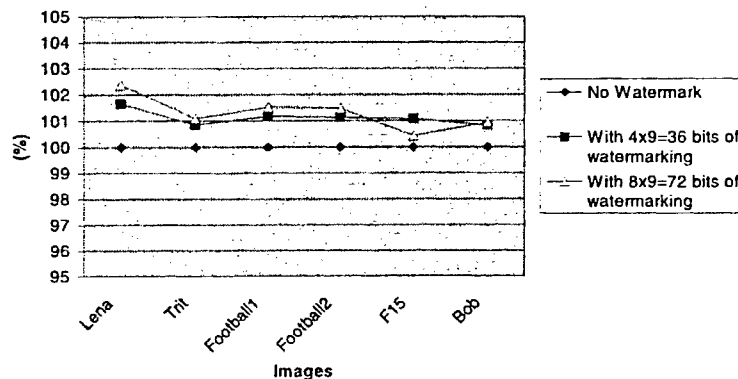
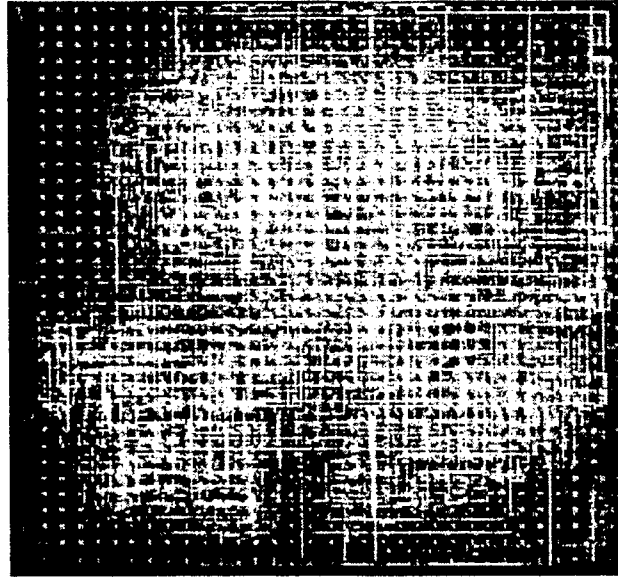


Figure 10. Cost of Watermarking the Huffman codes

Finally, the layout generated on an FPGA by synthesizing the Verilog code is shown in Figure 11. The critical path is 40ns without optimizations, and the clocking frequency is 25MHz. The empty CLBs (gray spots) can be used for embedding signatures as describe in [9,10].



**Figure 11. Layout from the Xilinx FloorPlanner for Linear Prediction
Coder Circuit**

Watermarking Method	Number of Signature	Cost
PDF manipulation for Huffman code	32 or 72	Maximally 2% inc. file size
Huffman code edge labeling	20 ~ 256 (theor. max)	None
Hardware Datapath Transformation	8 (4 pixels processed) ~??	No perform. loss (more gates)
Hardware Datapath Scheduling	6(1 bit per depend)	20% performance loss
State Machine	40 bits (5 bits per state)	More registers
FPGA physical layout	16 bits per CLB LUT	Reserved or unused CLBs

Table 3. Signatures and cost of signature for specified algorithm

Table 3 is a brief summary of all the bits embedded in the hierarchical design. The Huffman table is manipulated to embed either a 36 or 72 bit signature. The edge labeling embeds another 20 or more bits depending on the maximum code length for the error symbol with no negative impact. The transformations embed 8 bits of signature that can embed more bits if more image pixels are processed in parallel. The hardware scheduling decreases the performance due to the execution delay resulting from the artificial dependencies. More bits are embedded with more parallel execution. The finite state machine is used to embed 40 bits and the associated cost is more registers. The FPGA configurable logic blocks (CLB) are used to embed as many bits a possible in the FPGA physical layout.

5 CONCLUSION

Based on the results, we can cheaply and efficiently watermark a lossless linear prediction hardware using the Huffman table. The cost of adding a 72-bit signature in the Huffman code is at most 2% of the compressed image. The technique is simple enough to incorporate. Furthermore, the hardware methods allow for a "without reasonable doubt" conclusion that the design is that of the owners. The techniques used here are not limited to FPGAs and can be applied to ASICs. Given that it took some engineers a few weeks to reverse engineer an Intel 386 [11], it would be easy for others to copy one's hardware. By using the techniques developed here, one may embed a 20-bit signature. This 20-bit signature may be sufficient to prove in the court of law that the design is proprietary. Using 20-bits, there is a 1 in 1,000,000 chance that the next designer could use the same set of transformations, scheduling, and state machine watermarking. These efforts show that watermarking a lossless linear prediction for lossless image compression hardware and the Huffman table is simple and effective.

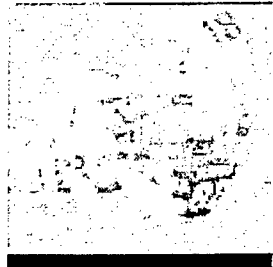
Appendix A: Images



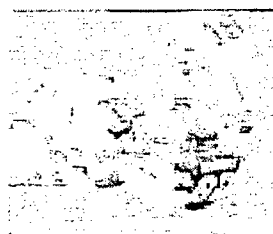
Bob



F15



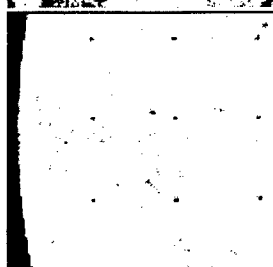
Fball1



Fball2



Lena



Trit

References

- [1] H.G. Mussman. Predictive Image Coding. *Advances in Electronics and Electron Physics, Supplement. 12*, pages 73-112. Academic Press Inc. 1979.
- [2] G.K. Wallace. The JPEG Still Picture Compression Standard. *Communications of the ACM*. 34(4):31-44, 1991.
- [3] M. Nelson and J-L. Gailly. *The Data Compression Book (second edition)*. M&T Books, 1996.
- [4] I.J. Cox, J. Kilian, T. Leighton, T. Shamoan, "Secure spread spectrum watermarking for images, audio and video", *International Conference on Image Processing*, 1996, vol. 3, pp. 243-246.
- [5] F. Hartung and B. Girod, "Watermarking of MPEG-2 encoded video without decoding and re-encoding", *Multimedia Computing and Networking*, 1997, pp. 264-274.
- [6] L. Boney, A.H. Tewfik, and K.N. Hamdy, "Digital watermarks for audio signals", *International Conference on Multimedia Computing and Systems*, 1996, pp. 473-480.
- [7] P. G Howard and J.S. Vitter. Fast and efficient lossless compression. *Proceedings of the Data Compression Conference*, pages 351-360. IEEE computer Society Press, 1993.
- [8] N. D. Memon, K. Sayood, and S. S. Magliveras. "Lossless Image Compression – a comparative study." *Still Image Compression*, pages 8-20. SPIE Proceedings Volume 2418, 1995.
- [9] J. Lach, W. H. Mangione-Smith, and M. Potkonjak. "FPGA Fingerprinting Techniques for Protecting Intellectual Property," *Custom Integrated Circuits Conference*, 1998.
- [10] J. Lach, W. H. Mangione-Smith, and M. Potkonjak. "Fingerprinting Digital Circuits On Programmable Hardware." *International Workshop in Information Hiding*, 1998.
- [11] R. Anderson and M Kuhn, "Tamper Resistance – A Cautionary Note", *USENIX Workshop on Electronic Commerce*, Nov. 1996.

On-line Fault Detection for Programmable Logic

Nathan R. Shnidman¹, William H. Mangione-Smith¹, and Miodrag Potkonjak²

Departments of Electrical Engineering¹ and Computer Science²

The University of California, Los Angeles

{naters, billms}@icsl.ucla.edu, miodrag@cs.ucla.edu

Abstract We introduce a technique for on-line built-in self-testing of Field Programmable Gate Arrays (FPGAs). This system detects deviations from the intended functionality of an FPGA without using special-purpose hardware or hardware external to the device, and without interrupting system operation. Such a system would be useful for mission-critical applications with resource constraints. The system solves these problems through an on-line fault scanning methodology. A device's internal resources are configured to test for faults. Testing scans across an FPGA, checking a section at a time. Simulation on a model FPGA supports the viability and effectiveness of such a system.

INTRODUCTION

The most common type of programmable logic device available today is the Field Programmable Gate Array (FPGA). As the name implies, FPGAs are made up of an array of programmable gates. The FPGA can be programmed by replacing the function of each gate and changing the connectivity among gates.

The use of FPGAs in consumer applications, in addition to their more traditional use in logic emulation systems, prototyping, and low volume applications, has increased due to recent advances in FPGA technology. Unfortunately, many of the trends that make newer FPGAs more appealing and affordable also make them less reliable. For example, high density programmable devices are made more susceptible to gamma particle radiation by smaller feature sizes, and the corresponding lower threshold voltages. Also, larger die sizes make interference from such radiation much more likely [1].

Ironically, while hardware reuse is a primary reason for utilizing an FPGA, external hardware for fault testing and tolerance of FPGAs often requires large amounts of additional system resources. Implementing these tasks external to the FPGA requires that the functionality of the device be interrupted in order to detect and address faults. This approach not only results in system functionality being interrupted, but also in fault testing only occurring periodically. Thus, there is likely to be a delay between the time when a fault occurs and when it is detected. Most importantly, soft faults in the programmable memory of an FPGA will not be detected by such a system without adding a time consuming step of reading the configuration out of the device.

Making use of resources internal to the FPGA to implement a fault detection system, however, avoids these problems. A portion of the device resources are set aside

to perform the fault handling, but fewer system resources are consumed than with an external fault monitor. An internal fault monitor can also run in the background on an FPGA that supports partial reconfigurability. Such a fault monitor could run continuously while not impeding device functionality. Another advantage to an internal fault monitor is that it conserves limited pin resources and avoids the relatively slow process of transferring information off-chip through the pins. This approach allows for rapid detection of both hard and soft faults.

The resources needed to perform fault testing can be kept to a minimum by using a fault scanning methodology. Only a small section of the FPGA is tested at a time, but continuous testing can scan across the FPGA assuring that the entire FPGA will be tested eventually. It is necessary to take the resources being tested offline (but not the entire system) to perform the fault detection tests. By testing only a small portion of the FPGA at a time the approach allows fault testing to occur without impeding functionality.

1.1. FPGA Architectures

FPGAs have cell-like structures. The cell is used to implement the functionality of a number of gates, and it also commonly contains a small amount of memory. The number of gates per cell is dependent on the FPGA architecture, but is usually anywhere from one to six gates. Part of the programmability of the FPGA comes from the fact that the designer can change the actual type of gates implemented by each cell. In addition, the user also determines if the combinational logic section of the cell or the memory section of the cell is used. It is sometimes possible to use both sections concurrently.

There are two basic models for the combinational logic portion of cells: island based and fine-grained [2]. The island based FPGA uses one or more look-up-tables (LUTs) per cell to provide the functionality of gates. These cells typically have four or more inputs. In contrast, a fine-grained FPGA cell usually has only two inputs. The small number of inputs often allows logic in a fine-grained cell to be implemented with multiplexers [3].

There are also two types of programmable interconnect. One type of interconnect involves point-to-point or segmented buses. This model has wires of varying lengths placed horizontally and vertically throughout the FPGA, with switches connecting the pieces of interconnect. Programming such an FPGA requires a routing step, where an attempt is made to connect cells together using the least amount of interconnect. Since interconnect needs are unknown *a priori* in such a model, it is possible that designs exceed the amount

Submitted to the IEEE Transactions on VLSI. An earlier version has been published at ARVLSI 1997 in Ann Arbor.

of available interconnect. A hierarchical structure is typically employed.

The other form of interconnect is termed bus-based. This model involves long interconnect lines which span all (or a significant portion) of the chip. Connections between cells are made by writing to and reading from these buses. Bus-based interconnect tends to be slower than point-to-point due to increased wire capacitance. However, bus-based interconnect has the advantage of predictable timing, because the time to drive all signals is the same.

To illustrate a commercial FPGA design, consider the Flex10K by Altera [4]. The Flex10K uses static memory based LUTs, with a single flip-flop (Figure 1). In addition to the basic cell structure, the Flex10K groups eight cells, or Logic Elements (LEs), into what is called a Logic Array Block (LAB). These LABs provide point-to-point interconnect on a small scale, for fast local communication. The LABs are arrayed in a grid pattern, and are connected by chip-length buses (Figure 2). The place-and-route software for the Flex10K attempts to constrain designs to units that fit within a LAB, and buses make routing after placement trivial.

1.2. Motivational Example

An on-line internal fault detection system would be of particular use in space-based applications [5]. Limited resources such as volume, weight, and power make the use of FPGAs particularly appealing due to the opportunity to use time-sharing among the circuits to increase functional density. Furthermore, the use of additional hardware to perform fault testing and fault tolerance is particularly unappealing because of the resource constraints. Thus, traditional approaches such as triple modular redundancy [1] are tolerated, but at a great expense. Space based systems are also subject to much more operational interference from radiation and charged particles than terrestrial systems. The probability of faults in a space situation is much higher. As such, it is imperative that faults be located and addressed quickly.

Also, in space based systems, proper functionality is often critical to the mission success. If an FPGA fails, it is vital to the mission that the fault be detected and handled as soon as possible. The system must be capable of autonomous and automatic fault detection and handling.

A fault scanning system would address most of these concerns. The fact that the fault scanner uses resources internal to the FPGA precludes an increase in the use of system resources. The transparent nature of the fault scanner also allows it to run continuously, thus providing quick detection of faults. If the fault tolerance mechanism discussed below is implemented, the fault scanner could even allow the FPGA to tolerate the fault and continue to function.

1.3. Paper Organization

This paper presents multiple possible internal on-line fault scanning monitors for FPGAs, and simulations showing the proof-of-concept for one implementation in particular. We will discuss the available design options and the simulation in Section 2. This section will also address the specific FPGA used for the simulation and how it compares to current FPGA models. We will discuss previous work in the area, and its implications on the work presented here in Section 3. The basic fault scanning system is developed in Section 4. Alternative faults scanners will be discussed in Section 5, along with the relative advantages of the various systems. Simulation results demonstrating the functionality of the fault scanner are presented in Section 6. Some concluding remarks are given in Section 7.

2. PRELIMINARIES

The assumptions used in implementing the fault scanning system will be discussed here. The types of faults addressed by this system are the Single-Stuck Fault (SSF) [6] and the Single-Event Upset (SEU) [6, 7]. This fault model is sufficient to verify LUTs and flip-flops. While the current design of the system does not yet address faults in the interconnect and control paths, other schemes have been proposed which address such faults [8-10].

It is reasonable to focus the current fault detection system on the LUTs and flip-flops, even though the majority of the FPGA area is taken up by interconnect. The fault detection system can only detect persistent faults, i.e. faults that will affect system operation until they are addressed. These types of faults are most likely to occur in the LUTs and flip-flops due to the fact that both of these elements have state.

Another assumption is that configuration memory and flip-flops are fault-free when the original configuration is loaded into the FPGA. Physical defects in the configuration elements, or errors in configuration data are not addressed by this testing system. A mechanism has been included, however, to help detect faults in the configuration elements and datapaths. The ability to read the configuration data which has been loaded into the FPGA could be used to detect such physical defects. The ability to read configurations after loading would provide a high level verification of the FPGA every time a new configuration is loaded.

Additionally, the FPGA cells must be modified as described below. It must be possible to mask the global control signals, so that control signals can target certain cells when necessary. Finally, the FPGA must be partially reconfigurable. Some FPGAs do not have the ability to change the functionality of only some cells, while leaving the remaining cells untouched. The ability to change the functionality of only a section of the FPGA is at the heart of this on-line fault system, and is absolutely necessary to implement the system. However, it is not essential that the system present a partial reconfiguration capability to the designer.

3. RELATED WORK

The related work falls into four general categories: built-in self-test (BIST), built-in self-repair (BISR), FPGA yield enhancement, and FPGA faults in space based systems.

BIST and BISR methods for detecting and handling faults have been used extensively in memory designs [6, 11, 12], as well as in FPGAs in an offline manner [8, 13-18]. The FPGA BIST techniques mainly focus on configuring the entire FPGA into known states in order to identify faults. While these techniques work and are well suited for certain applications, they all require configuration of the entire FPGA, and therefore interrupt system operation. BIST techniques can be combined with BISR and other fault tolerance methods [19, 20] to increase the robustness of FPGAs.

BIST and even BISR methods have been used to detect and handle fabrication faults in order to improve fabrication yields [21-24]. There has also been work done dealing specifically with the susceptibility of FPGAs to faults in space based situations [25, 26].

None of the above work, however, deals with on-line fault detection. Some of these systems provide fault detection, using BIST, on FPGAs. The major difference between our fault scanner system and previous work is that with the new approach the FPGA need not be taken offline before fault testing can occur, i.e. the functionality of the system is not interrupted for testing purposes.

4. APPROACH

We present a system overview, the basic algorithm, and a discussion of the general FPGA architecture used in testing in this section.

4.1. System Overview

The basis of the internal FPGA fault system is a scanning methodology. The system allocates a portion of the FPGA to fault testing. Testing is accomplished by sweeping the test functions across the entire FPGA. If the functionality of a small number of FPGA elements can be replicated on another portion of the FPGA, then those components can be taken off-line and tested for faults in a transparent manner (i.e. without interrupting functionality). The fault scanning system can then move on to another set of elements to copy and then test, thus systematically moving throughout the entire FPGA testing for faults.

4.2. Basic Algorithm

Built in fault testing for FPGAs requires that some of the resources of the FPGA be used for testing purposes. Allocating too many resources reduces the functionality of the FPGA, while allocating too few resources results in slow testing. The basic unit of testing is a column. Focusing on a column at a time allows for parallel testing

of multiple cells, while not excessively constraining the FPGA functionality.

Our basic on-line testing algorithm reserves two columns of the FPGA for testing. One of these columns, the Testing Column (TC), contains the testing state machine. This state machine produces the control signals that implement testing. If the state machine is too large for a column in an FPGA architecture, then the state machine must be run off-chip, with the control signals as chip inputs. At the moment, a second state machine, which keeps track of the column being tested, is modeled as being off-chip. For designs of 20 cells per column or larger, this state machine could also be implemented on-chip. The output of this state machine is used to mask the global testing control signals such that they only affect a single column. The other reserved column is the Free Column (FC), which acts as a buffer space during testing.

The on-line testing algorithm consists of three basic steps: copy, test, and move (Figure 3). In the copy step, a functional duplicate is made of the next column to be tested. Copying is done by writing the data from the configuration memory and the configuration flip-flop to the configuration data (CDATA) bus (Figure 4). This data on the CDATA bus is written to the FC, thus making a functional duplicate of the column to be tested. Since only a single LUT bit can be accessed at a time, both the configuration memory and the FC LUTs are sequenced through all possible input vectors in parallel, by connecting their inputs to a counter.

After the configuration memory is copied, the inputs and outputs of the FC are switched over to those of the cells in the column to be tested. The bus-based architecture of the FPGA makes this relatively simple. The FC cells simply tap the input buses of the cells being duplicated and the information in the configuration memory is used to set the outputs to drive the appropriate buses.

Next, the values in the flip-flops in the column to be tested are sent over the CDATA bus. These values are written to the FC flip-flops. If there is a write to a flip-flop in the column to be tested, while it is being copied, the write always wins over the copy. This priority ensures that outdated values are not copied into the FC flip-flops. If a write to a flip-flop of a cell in the column to be tested occurs during the copying process, the new value is also written into the flip-flop of the corresponding cell in the FC (because both flip-flops have the same inputs). Once the column to be tested has been copied, the FC outputs are turned on. Both columns are active with the same inputs, outputs, and functionality for a clock cycle in order to avoid glitches on the outputs. The outputs from the column to be tested are then tri-stated.

The next step is to perform the actual testing. The inputs to the column under test are connected to the output of the counter that is driving the configuration memory. The functions of the LUT and configuration memory are sequenced in parallel. Any difference between the output of the LUT and memory of any cell being tested indicates a fault. The outputs of the flip-flops and configuration flip-

flops are then compared. Any difference between those also indicates a fault. These procedures test for both SSF and SEU faults. It is necessary to have two copies of the correct cell functionality (i.e. the LUT and configuration memory and the two flip-flops) in order to detect SEU faults.

The next phase of testing is to write the inverse of the values in the configuration memories and configuration flip-flops to the LUTs and flip-flops, thus allowing the system to check for SSF faults. The outputs of the LUTs are then compared to the inverted outputs of the configuration memory to test for differences. Any discrepancy between the outputs indicates a fault. The outputs of the flip-flops are compared in a similar fashion to the inverted outputs of the configuration memories. It is important to note that, while testing a cell for faults, it is possible to write to the cell's flip-flop independently of writing to the configuration flip-flop, although the converse is not true.

Once the testing is completed, the move step begins. This step involves transferring the original functionality back into the column that was just tested. This phase begins by writing the non-inverted values in the configuration memories back into the LUTs. The inputs to the LUTs are then switched back to the correct buses. Next, the values stored in the FC flip-flops are written back to both the original flip-flops and configuration flip-flops. Writes take priority over copies, as with the previous copying to the FC flip-flops.

Finally, the outputs of the column that was just tested are turned on, and after waiting an extra clock cycle to avoid glitches, and the FC outputs are tri-stated. The algorithm is then applied to the column to the right of the column that was just tested. If there is no column to the right, testing begins at the leftmost column. This algorithm can be applied to both the TC and FC themselves, thus testing the entire FPGA.

4.3. System Architecture

The basic cell configuration can be seen in Figure 4. The main functionality of the cell consists of a four-input, one-output memory based LUT, and a single flip-flop. The LUT is simply a static memory with the four address bits used as the inputs to the LUT. The values in the LUT are stored during configuration of the FPGA. The cell has multiple possible functional modes that utilize the main cell elements differently: LUT alone, flip-flop alone, flip-flop controlled by LUT inputs, and LUT and flip-flop together.

In addition to functionality similar to that of a typical FPGA cell, the cell in Figure 3 has extra elements that allow for fault testing. The major changes to the design are the addition of the configuration memory and the configuration flip-flop. In order to copy a cell, the LUT must be sequenced through all possible inputs, and the results written to the corresponding FC cell. The flip-flop data must also be copied. These elements allow the

configuration data of a specific cell to be accessed without affecting the operation of that cell. The configuration elements are also used in fault detection by comparing their output with the LUT and flip-flop outputs. It is important to note that, during normal functionality, all writes to a cell's flip-flop write the same data, simultaneously, to the configuration flip-flop. This step guarantees that the data in the configuration flip-flop is updated with valid data.

The cells are arrayed in a grid pattern in the FPGA and are connected by device-length buses (Figure 5). Horizontal buses carry the cell inputs, which are 4 bits wide, and the vertical buses carry the cell outputs, which are 1 bit wide. There are as many horizontal buses associated with each row of cells as there are cells in a row, and as many vertical buses as there are cells in a column. A bus also exists for the output of each cell in that column.

Switches connect each vertical bus to every horizontal bus in the FPGA. For example, if the output of cell (1,1) is to be used as the LSB and MSB inputs into cell (2,3) the switches connecting the vertical output bus of cell (1,1) to the first and fourth bits of the input bus to cell (2,3) would be turned on. The state of these switches is set by the configuration data loaded into the FPGA. A copy of the state loaded into all of the switches at a juncture is also stored in the configuration memory at that juncture. So the state of all switches at the juncture of row 3 and column 2 are stored in the configuration memory of cell (2,3). The switches have the ability to write their state value to the CDATA bus (see below). Thus, during the testing phase the state of each switch at a juncture can be compared with its intended value, which is stored in the configuration memory. Any discrepancy between the two values signals a fault. Additionally, this capability allows switch settings to be copied to the FC during the copy phase.

In addition to the cell-to-cell connections, there are also global connections within the FPGA. Many of the global connections carry control signals to the cells. Global signals can be masked to affect only specific cells, or columns of cells, in addition to all cells.

Most of the global connections not used for control signals are used to carry configuration data in the LUTs and flip-flops. Of particular interest is the CDATA bus. There is a CDATA bus for each row in the FPGA. The bus is used to access the configuration data of a cell so that a copy of that cell can be made. This capability is necessary for on-line fault testing to occur.

The other configuration data global connections are used whenever a new configuration is input into the FPGA. It should be noted, however, that it is possible to eliminate the global nature of these configuration connections by simply tying them to the CDATA bus. This would provide the same functionality, but would theoretically slow down the reconfiguration process, as there is only one CDATA bus per row. Thus, configuration data would have to be input sequentially by cell, in each row, instead of just configuring all cells in parallel. However, since I/O pins on an FPGA are usually limited, configuring all cells in parallel is generally

# of cells in Column	Scanning Clock Frequency					
	100MHz	50MHz	25MHz	12.5MHz	6.25 MHz	3.125 MHz
16 Cells	71839	35920	17960	8980	4490	2245
32 Cells	35920	17960	8980	4490	2245	1123
64 Cells	17960	8980	4490	2245	1123	561
128 Cells	8980	4490	2245	1123	561	281

Table 1: Number of Scans of an FPGA in a second

not possible, thus making the elimination of the extra connections preferable.

The only other type of global connection is the Fault Bus (FB). The FB is similar to the CDATE bus in that there is one such bus per row, which stretches the entire length of the FPGA. Each cell in a row is connected to the FB, but the fault output of each these cells is tri-stated if that cell is not currently being tested. The use of the FB in fault notification will be discussed later.

The simulated FGPA was loosely based upon the Altera Flex10K part. The main similarities consisted of the use of chip-wide buses as interconnect, and the use of four-input, one-output lookup tables in the cells. The simulated FPGA, however, does not make use of grouped cells (LABs) and dedicated memory blocks as does the Flex10K.

4.4. Testing Issues

An important aspect of the fault scanner is that the time between subsequent scans of any cell is relatively low, thus reducing operating time in a faulty mode. The time between consecutive scans of a given cell is:

$$\tau_{scan} = [7 * (clk) + 5 * (2^l * clk)] * N$$

where clk is the scanning clock period, l is the number of input bits to each LUT, N is the number of columns in the FPGA, and the constants seven and five represent the number of steps of each length in the testing algorithm. The longer step size (i.e. $2^l * clk$) represents steps in which the functionality of a LUT must be sequenced through. It should be noted that the scanning clock period may be a multiple of the system clock. Figure 6 shows the number of scans per second for an FPGA with 4-input LUTs as the clk period and number of columns varies, and the numbers represented in the figure are shown in Table 1.

4.5. Fault Identification

An I/O pin is used to indicate that a fault has occurred. The input to this pin is the logic OR of the data on all of the FBs. If a fault is detected, the entire FPGA is reconfigured to some default state. If the fault was an SEU, then reconfiguration will fix the fault and operation restarts. If a fault in the same cell persists, then it is most likely a SSF. The only way to fix such a fault, without replacing the FPGA, is to avoid using that cell.

The first step in avoiding the use of the faulty cell is to identify which cell contains the fault. When a fault occurs FPGA functionality is stopped. Many FPGAs provide the ability to read the configuration after loading it, in order to support debugging. This feature allows the internal state of the FPGA to be viewed externally when the FPGA is inactive. With this approach it is possible determined which FB indicated a fault. That FB corresponds to a specific row. The state stored in the state machine that keeps track of the current column being tested denotes in which column the fault occurred. Once the row and column of the faulty cell have been determined the fault has been identified. The fault can then be handled by a number of fault tolerance schemes. A possible fault tolerance scheme is discussed in a later section.

5. ALTERNATE IMPLEMENTATIONS

In addition to the testing scheme described above, there are multiple variations on this scheme which could be implemented.

5.1. Multiple Column Testing

It is possible to speed up the time it takes to scan the entire FPGA by having multiple FCs. This would allow scanning of multiple sections of the chip in parallel. The FPGA could be partitioned into sections with an equal number of columns in each section. Each section would have its own FC, and the CDATE and FB for each section would be independent. The CDATE buses must be segmented so that all sections can transfer cell data to their respective FC concurrently. The FB must be segmented so that faults can be detected independently in each section, and so that there is no contention on the buses. Since the control signals from the TC are distributed on a masked version of the global control signals, it is possible to implement this scheme using only a single TC. If the sections do not contain exactly the same number of columns, however, the current column to be tested must be stored separately for each section width.

This scheme has the obvious disadvantage of requiring more of the FPGA resources to be dedicated to testing purposes. However, it will increase the speed of testing for faults by approximately the number of sections. The percentage overhead for varying number of TCs on multiple sizes of FPGAs can be seen in Table 2.

# of cells / Column	% Overhead		
	1 Scanner	2 Scanners	3 Scanners
16 Cells	12.50%	25.00%	37.50%
32 Cells	6.25%	12.50%	18.75%
64 Cells	3.13%	6.25%	9.38%
128 Cells	1.56%	3.13%	4.69%

Table 2 : Resource overhead for scanning system vs. number of testing columns used for concurrent scanning

5.2. Moving Free Column

One alternate implementation of fault scanning would be to have a moving FC that scanned across the FPGA following the column to be tested. This would require copying the functional of each column to be tested no further than an adjacent column. After each column is tested it becomes the new FC. This change avoids the two steps in the testing scheme which write the original configuration back into the column which was testing, and then the one extra step of having a handoff to bring the tested column back on-line. This modification decreases the time to scan a chip to:

$$\tau_{scan} = [5 * (clk) + 4 * (2' * clk)] * N$$

Another advantage of such a scheme is that all columns are functionally identical. In the original scheme the FC needed extra capability at its input and output to allow it to connect to the inputs and outputs of any other column. In this scheme, a single cell and interconnect model can be used to create the entire chip, which simplifies the design.

There are some disadvantages to this scheme. The first of these is that, while the interconnect associated with copying a column is localized, and therefore decreased, control interconnect is still global. The TC does not move, and as such, the control signals must travel across the chip. This long interconnect could limit the scanning speed of the FPGA. This issue will be addressed in the next section.

This approach also results in a variation in time between testing of each column. The original scheme simply swept from one side of the FPGA to the other, and then restarted at the beginning. Thus, the time between testing passes of any column is always the same. This modified scheme has to scan back and forth across the chip, so the time between passes of a given column alternates depending on the scanning direction (Figure 7).

A moving FC scheme could be implemented on a bus-based FPGA. This approach would provide the advantage of identical columns, but at the expense of increased resources. Instead of expanding the input and output capabilities of a single column, as in the current scheme, the input and output capabilities of every column would have to be expanded to allow each column to function as its neighbor.

5.3. Moving Testing and Free Columns

Another option is to have both the FC and the TC move together. This approach removes the problem of having the control signals travelling across the chip. Using a moving TC means that local interconnect could be used to distribute control signals. There are other issues that arise, however. First, cell-to-cell interconnections must be increased by one column more than in the scheme where only the FC moved, so that signals can be passed to columns across the TC and FC. Second, the extra algorithm steps saved in switching to a moving FC are replaced by steps to move the TC. It is still not necessary to re-implement a column's function after it has been tested, but the column that was just tested now becomes host for the TC.

5.4. Multiple Testing Columns

Another possible implementation variation is to have multiple TCs with the above scheme. That is, to have a similar scenario as describe in the Multiple Column Testing case, but with a dedicated TC for each section. This would decrease the amount of time between scans for each column, as in the Multiple Column Testing case, but would require an increase of two times the number of sections in resource usage for testing purposes. A major advantage of this approach over that of the Multiple Column Testing case is that the network of control signals can be partitioned for each section. This would allow control signals to arrive slightly more quickly. This case has the disadvantages, however, of requiring an extra column for each new TC, and of requiring the segmentation of the control signal interconnect *a priori* in order to take advantage of the multiple TCs.

5.5. Point-To-Point Interconnect

Many FPGAs utilize point-to-point (or segmented) interconnect for routing. It is possible to implement fault scanning on such FPGAs, but with some modifications.

The Xilinx 4K family of parts is an example of a FPGA family which utilizes a hierarchical segmented network. Segmented interconnect consists of wires which make specific point-to-point connections. Each cell has multiple wires available each of which connects that cell to a specific other cell.

In the Xilinx 4K parts these wires are of specified lengths. There are wires to connect to cells 1, 2, 4, and 16 cells away. Thus a connection can be made to a cell which is five cells away by routing it through an intermediate cell using the 4-cell wire and then making use of the nearest-neighbor connection of that intermediate cell to route to the desired destination cell.

In order to implement fault scanning on an FPGA using segmented interconnects the ability to forward information to and from a column would be necessary. This would allow a FC to share the same input and output connections as the column it is mimicking. Any inputs or outputs to the column being tested could simply be routed the extra distance to the copy of that column. This forwarding of data would need to

be taken into account during the place-and-route and simulation steps, as it means that the delay associated with a given connection can change slightly over time.

Another necessary modification would be to the FC and TC. Instead of having a stationary FC and TC, the FC and TC would migrate across the FPGA, as in Section 5.3. Having the FC and TC localized to the column being tested minimizes the amount of interconnect necessary. Minimizing interconnect is necessary because interconnect takes up the majority of the area in the FPGA and can thus be a limiting factor in FPGA design. Longer interconnect also increases the amount of time for testing and can limit the scanning speed of the FPGA. A migrating FC and TC scheme could make use of 2-cell distance wires to forward information to and from the new copy of the column being tested, and the nearest-neighbor, or 1-cell distance, wires to perform testing.

The placement of the configuration memory for a given column would also need to be changed for such a scheme if the FPGA interconnect is segmented. Each configuration memory services two columns, one for each possible copy of the cell configuration stored in the configuration memory. Placing the configuration memory in between these two columns equalizes the interconnect needed to connect the configuration memory to each column. Such a placement of the configuration memory also allows for the use of nearest-neighbor interconnect for testing purposes, leaving the 2-cell length interconnect free for data forwarding purposes. This helps facilitate the transfer of configuration data without interrupting the normal function of a cell.

5.6. Fault Tolerance

It is possible to increase the capability of the fault scanning system to include fault tolerance as follows. If a fault is found in a cell, the cell's configuration is loaded into the FC cell on the same row as the faulty cell. If the fault is determined to be an SSF fault, then the FC cell can be switched to the inputs and outputs of the faulty cell and the faulty cell itself can simply be taken offline. This method of fault tolerance can only accommodate a single fault in each row for each FC. An approach similar to this one is used during manufacturing test for the Altera Flex10K, though resources are switched through OTP fuses in manufacturing.

6. EVALUATION

Simulation and testing was originally done using the Cadence toolset (HDL desktop, Leapfrog VHDL simulator, and Waveview wave display) on a Sparc20. Later, simulation was moved to a Pentium Pro making use of the V-System simulator by ModelTech.

An FPGA simulation has been implemented in VHDL. A PERL script generates some of the VHDL files, allowing the simulation to be independent of the size of the FPGA. This script takes as input the number of rows and columns

in the FPGA to be simulated, and generates VHDL for an FPGA of the specified size.

In addition to the VHDL code, the simulation makes use of six files that specify control signal and input data. These files provide external input to the simulated FPGA.

The FPGA simulated here has 4 rows and 4 columns. A 4x4 array is large enough to implement non-trivial functions in the two active columns and allow scanning to be tested, but is small enough that creating configuration data by hand is feasible.

The scanning clock rate for the simulated FPGA was 25 MHz, a reasonable rate for existing FPGAs. The scanning clock rate should be the highest possible multiple of the clock rate of the FPGA. This will allow scanning to proceed as fast as possible, while not requiring excessive amounts of resources to implement multiple clocks or synchronization of scanning with the operation of the rest of the FPGA.

Two different designs, each requiring four cells, were implemented simultaneously on the FPGA (Figure 8). One design utilized only the combinational logic features of the cells to implement two comparators. Each comparator took as input two three-bit words, A and B, and tested if $A > B$. Two comparators were implemented so that twice as many input vectors could be tested in a given period of time. Each comparator required two cells, spaced out over two adjacent columns.

The second design was a state machine that made use of both flip-flops and LUTs. The state machine itself was a simple design requiring only the remaining four cells. Three of the cells held state information, while the fourth implemented logic based on the current state. The state machine took as its inputs the system clock and a state machine reset signal. Unlike the comparators, the state machine implemented functionality across row boundaries in addition to column boundaries.

The simulation demonstrates that the on-line fault testing system is feasible. It is critical to void glitches during the hand-off between the FC and column being tested, both when the control is given to the FC and when it is returned to the column being tested. Glitches are avoided by driving both columns with the same inputs, and enabling both columns' outputs to drive the same bus during the hand-off. Since the columns are configured identically, this technique results in the same output being driven onto the bus by both columns. The column to be taken off-line can then be disabled, and the other column takes over providing the function. This process is shown in Figure 9, which displays the output signal for row 1 while control is being passed from the column being tested, column 2, to the FC, column 1. The dashed line in the figure represents tri-stating. Control is passed seamlessly, without any glitches on the output. The output shown here is from one of the comparator circuits.

A second critical system property is the assurance that writes to the flip-flops always occur properly. It is vital that no writes be lost during the copying process. A mechanism of having writes take priority over the copy ensures that flip-flop values are copied as necessary, but that a stale value is

Submitted to the IEEE Transactions on VLSI. An earlier version has been published at ARVLSI 1997 in Ann Arbor.

never written. Figure 10 shows a flip-flop write taking place during the copy operation, and shows that the correct value is written and produced by the flip-flop. There is a delay between writes and a change in the flip-flop output because the system is falling-edge triggered, while the flip-flops are rising-edge triggered.

A last critical system element is that faults be properly detected. Figure 11 shows the discovery of an induced fault in the flip-flop element of a tested cell. The system accurately detects faults in both LUTs and flip-flops.

7. CONCLUSION

The ability of reconfigurable systems to self-diagnose on-line is important to the viability of their use in many environments. Techniques for on-line fault identification and fault tolerance in FPGA systems have been presented here. Such techniques provide the assurance of proper device functionality in a continuous manner with low hardware overhead. This capability allows faults to be identified and handled as quickly as possible, in the least intrusive manner possible. The multiple different fault detection techniques allow a tailoring of the fault monitor used to the system on which the monitor will be implemented.

A simulation has been created in order to prove the feasibility of such techniques. The simulation shows that fault detection can occur without affecting device functionality, and at a high enough rate to ensure an appreciably small amount of time between a fault's occurrence and its detection.

REFERENCES

- [1] D. P. Siewiorek and R. S. Swartz, *Reliable Computer Systems: Design and Evaluation*. Burlington, MA: Digital Press, 1992.
- [2] S. Trimberger, K. Duong, and B. Conn, "Architecture Issues and Solutions for a High-Capacity FPGA," *FPGA97*, Monterey, CA, 1997.
- [3] J. Rose, R. Francis, D. Lewis, and P. Chow, "Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency," *IEEE Journal of Solid State Circuit*, vol. 25, pp. 1217-1225, 1990.
- [4] Altera, *Data Book*. San Jose, CA: Altera, 1996.
- [5] K. W. Bernhardt, "Advanced Technologies for a Command and Data Handling Subsystem in a 'Better, Faster, Cheaper' Environment," 14th DASC Digital Avionics Systems Conference, Cambridge, MA, 1995.
- [6] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Designs*. New York: Computer Science Press, 1990.
- [7] J. P. Hayes, "On Modifying Logic Networks to Improve Their Diagnosability," *IEEE Transactions on Computers*, vol. 23, pp. 56-62, 1974.
- [8] H. Michinishi, et al., "A Test Methodology for Interconnect Structures of LUT-Based FPGAs," the Fifth Asian Test Symposium, Los Alamitos, CA, 1996.
- [9] W. K. Huang, X. T. Chen, and F. Lombardi, "On the Diagnosis of Programmable Interconnect Systems: Theory and Application," 14th IEEE VLSI Test Symposium, Los Alamitos, CA, 1996.
- [10] T. Liu, F. Lombardi, S. Horiguchi, and J. H. Kim, "A Structured Walking-1 Approach for the Diagnosis of Interconnects and FPGAs," *IEICE Transactions on Information and Systems*, vol. E79-D, pp. 29-40, 1996.
- [11] K. N. Levitt, M. W. Green, and J. Goldberg, "A Study of the Data Communication Problems in Self-Repairable Multiprocessors," AFIPS, Washington, D. C., 1968.
- [12] V. D. Agrawal, C. R. Kime, and K. K. Saluja, "A Tutorial on Built-in Self-Test," *IEEE Design and Test of Computers*, vol. 10, pp. 69-77, 1993.
- [13] C. Stroud, S. Konala, P. Chen, and M. Abramovici, "Built-In Self-Test of Logic Blocks in FPGAs (Finally, a Free Lunch: BIST Without Overhead!)," the 14th IEEE VLSI Test Symposium, 1996.
- [14] W. K. Huang and F. Lombardi, "An Approach for Testing Programmable/Configurable Field Programmable Gate Arrays," 14th IEEE VLSI Test Symposium, 1996.
- [15] T. Inoue, et al., "Universal Test Complexity of Field-Programmable Gate Arrays," Fourth Asian Test Symposium, Los Alamitos, CA, 1995.
- [16] X. T. Chen, W. K. Huang, F. Lombardi, and X. Sun, "A Row-Based FPGA for Single and Multiple Stuck-At Fault Detection," IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems, Lafayette, LA, 1995.
- [17] C. Jordan and W. P. Marnane, "Incoming Inspection of FPGAs," Third European Test Conference, Los Alamitos, CA, 1993.
- [18] K. Kwiat, W. Debany, and S. Hariri, "Effects of Technology Mapping on Fault-Detection Coverage in Reprogrammable FPGAs," IEE Proceeding-Computers and Digital Techniques, 1995.
- [19] G. A. Mojoli, et al., "KITE: A Behavioral Approach to Fault-Tolerance in FPGA-Based Systems," IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Boston, MA, 1996.
- [20] R. Cuddapah and M. Corba, "Reconfigurable Logic for Fault Tolerance," 5th International Workshop on Field Programmable Logic and Applications, Oxford, UK, 1995.
- [21] F. Hancsek and S. Dutt, "Node-Covering Based Defect and Fault-Tolerance methods for Increased Yield in FPGAs," the Ninth International Conference on VLSI Design, 1995.
- [22] N. J. Howard, A. M. Tyrrell, and N. M. Allinson, "The Yield Enhancement of Field-Programmable Gate Arrays," *IEEE Transactions on VLSI Systems*, vol. 2, pp. 115-123, 1994.
- [23] K. Roy and S. Nag, "On Routability for FPGAs Under Faulty Conditions," *IEEE Transactions on Computers*, vol. 44, pp. 1296-1305, 1996.
- [24] J. L. Kelly and P. A. Ivey, "Defect tolerant SRAM based FPGAs," *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 479-482, 1994.
- [25] G. Swift and R. Katz, "An Experimental Survey of Heavy Ion Induced Dielectric Rupture in Actel Field Programmable Gate Arrays (FPGAs)," *IEEE Transactions on Nuclear Science*, vol. 43, pp. 967-972, 1996.
- [26] K. A. LaBel, A. K. Moran, D. K. Hawkins, J. A. Cooley, and e. al., "Single Event Effect Proton and Heavy Ion Test Results for Candidate Spacecraft Electronics," *IEEE Radiation Effects Data Workshop*, pp. 64-71, 1994.

Submitted to the *IEEE Transactions on VLSI*. An earlier version has been published at *ARVLSI 1997* in Ann Arbor.

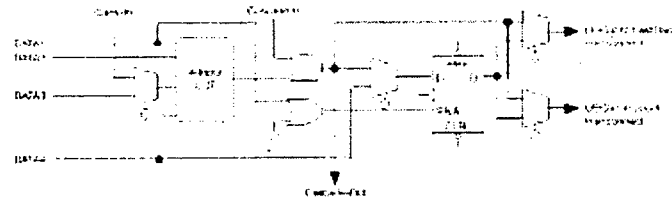


Figure 1: Flex10K cell [4]

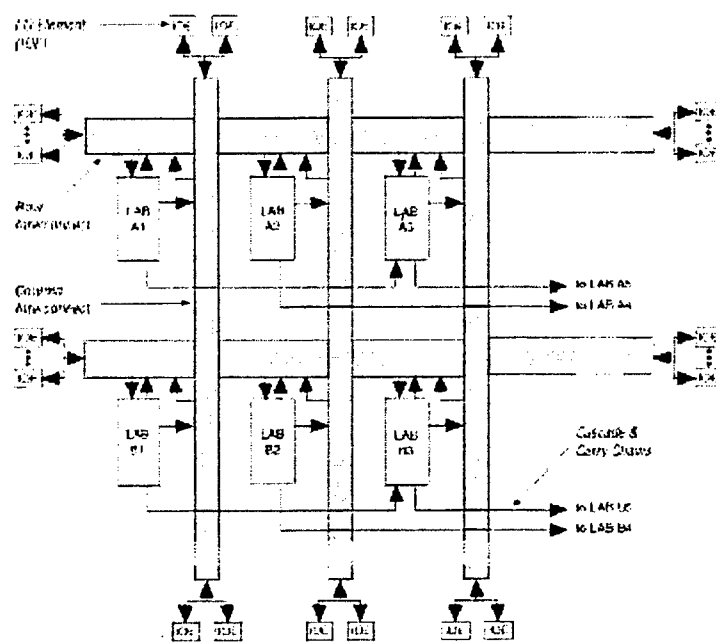


Figure 2: Flex10k Array [4]

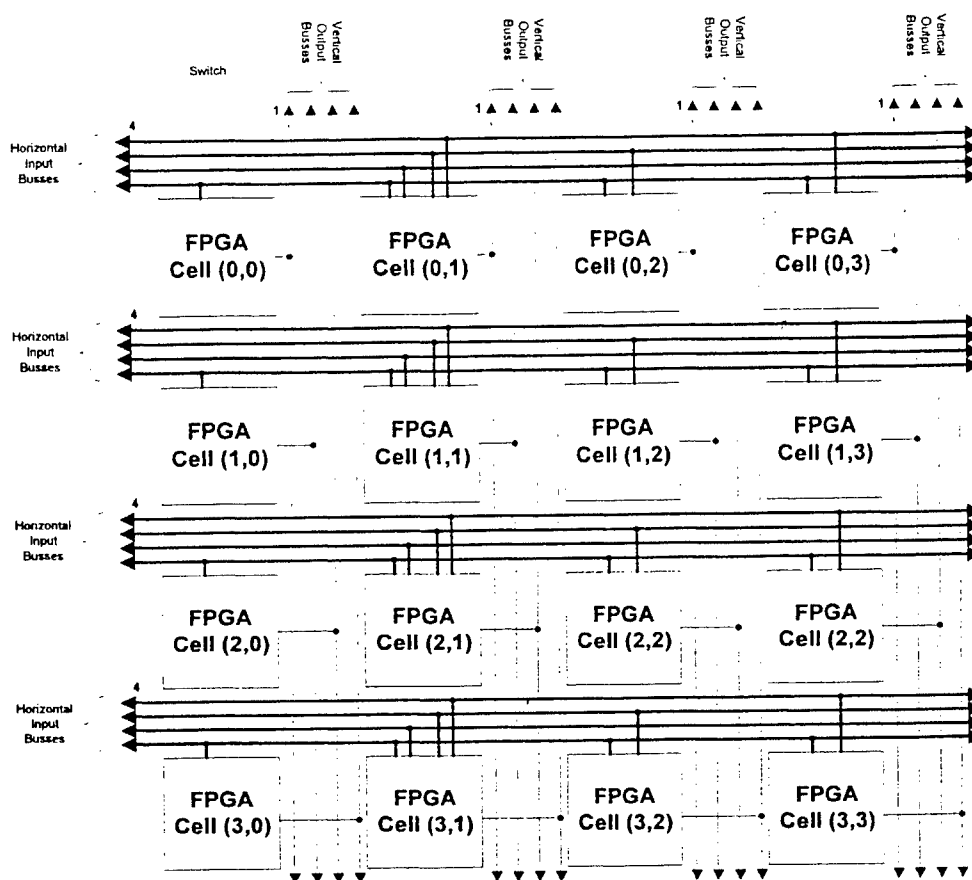


Figure 5: FPGA Cell Array with Interconnect

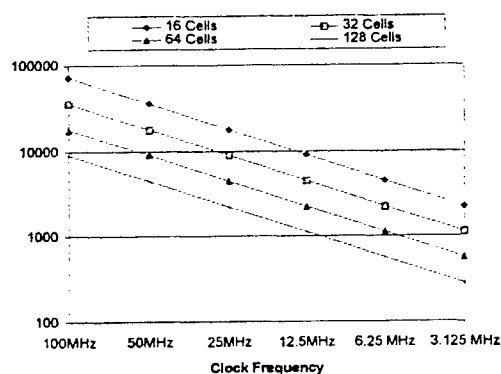


Figure 6: Number of times an FPGA of a given size can be scanned in one second, for multiple scanning clock frequencies

Submitted to the IEEE Transactions on VLSI. An earlier version has been published at ARVLSI 1997 in Ann Arbor.

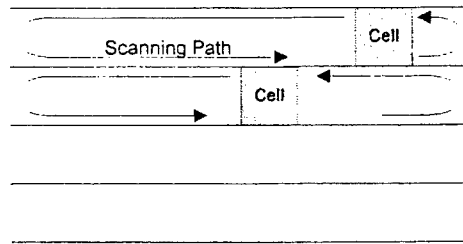


Figure 7: Scanning routes for moving FC and TC

Testing Cell	Free Cell	State Machine Cell 1	State Machine Cell 2
Testing Cell	Free Cell	Comp 1 Cell 1	Comp 1 Cell 2
Testing Cell	Free Cell	Comp 2 Cell 1	Comp 2 Cell 2
Testing Cell	Free Cell	State Machine Cell 3	State Machine Cell 4

Figure 8: Simulation Cell Functionality

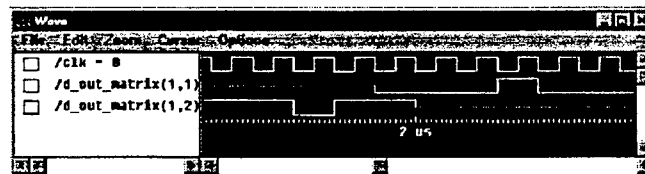


Figure 9: Waveform of hand-off of functionality from column 2 to the Free Column (column 1)

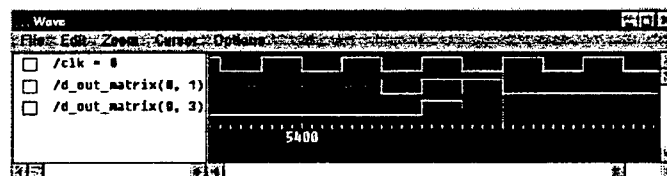


Figure 10: A write taking place during a copy operation

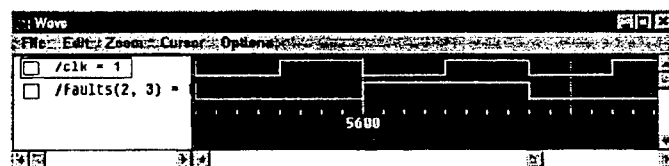


Figure 11: A fault is detected in the LUT of cell (1,3)

A Case Study of Partially Evaluated Hardware Circuits: Key-Specific DES

Jason Leonard and William H. Mangione-Smith
Electrical Engineering Department, University of California
Los Angeles, CA 90095-1594
<http://www.icsl.ucla.edu/~billms>

Abstract. FPGA based data encryption provides greater flexibility than ASICs and higher performance than software. Because FPGAs can be reprogrammed, they allow a single integrated circuit to efficiently implement multiple encryption algorithms. Furthermore, the ability to program FPGAs at runtime can be used to improve the performance through dynamic optimization. This paper describes the application of partial evaluation to an implementation of the Data Encryption Standard (DES). Each end user of a DES session shares a secret key, and this knowledge can be used to improve circuit performance. Key-specific encryption circuits require fewer resources and have shorter critical paths than the completely general design. By applying partial evaluation to DES on a Xilinx XC4000 series device we have reduced the CLB usage by 45% and improved the encryption bandwidth by 35%.

1. Introduction

As the volume of digital communications increases, reliable encryption becomes more important in order to ensure secure communications. In 1977 the U. S. National Bureau of Standards certified the Data Encryption Standard (DES) for use within the United States [1]. We will present a configurable computing implementation of DES.

Configurable computing offers a great degree of flexibility in system design due to the programmability of FPGAs. Circuit resources can be used for multiple purposes and refined to perform highly specific functions, thus combining generality with high performance. Secure communications systems often require the capacity to encrypt messages with several different algorithms in addition to the need to change keys regularly. ASICs lack the flexibility to implement multiple encryption algorithms efficiently. On the other hand, software implementations suffer from low speed. Fortunately, certain properties shared by many encryption algorithms make them well suited to FPGA implementations [2]. In particular, the reliance on many parallel bit-level operations generally matches the strengths of many FPGA architectures. However, the implementation of some other common functions, in particular wide permutations, is often problematic on FPGA architectures. This paper will present one technique for improving the performance of an encryption engine.

Encryption algorithms require a significant amount of hardware resources due to their inherent complexity. Our approach alleviates part of this burden by generating key-specific circuitry. This technique has the benefit of improving the speed of the circuit, as FPGA performance is strongly effected by routing complexity. The approach is termed partial evaluation; the effects of the encryption key are evaluated once rather than continuously during execution. FPGA configuration time is often not a problem as encryption keys change at a lower rate than encryption operations.

Session-based configuration has some additional advantages as well. In particular, it is more difficult for a third party to compromise the system integrity. With an ASIC implementation, an attack could involve writing a new value into the

key register. For the FPGA implementation developed here such an attack would involve replacing an entire FPGA configuration, which would be difficult to do unobtrusively. There has also been much discussion in the cryptography community about the security of DES because of the 56-bit key length. A commonly presented solution to present or future attacks on DES is to increase the key length. To go to an expanded DES key in an ASIC requires a new design. In a configurable design, however, it is possible to reuse the same physical devices.

The remainder of this paper is organized as follows. Section 2 discusses the relevant background work. Section 3 describes the DES algorithm in detail. Section 4 presents the approach used for specializing DES to a particular encryption key. Section 5 compares the resource requirements and performance of the full DES algorithm against the specialized implementation using partial evaluation. Finally, section 6 discusses some technology issues related to implementing this approach.

2. Related Work

The use of FPGAs for DES has attracted attention primarily in code-breaking machines, through the use of a "known plaintext" attack. These attacks require the possession of ciphertext and a part of the corresponding plaintext. Known plaintext attacks search the entire set of valid keys and hunt for a key which maps some known plaintext to some known ciphertext. FPGAs have received attention for this application because, while being slower than custom hardware, their cost is generally much lower and they outperform software approaches [3-5]. A large number of FPGAs can therefore form a system that searches the key-space in parallel. ASIC implementations suffer in comparison due to high non-recurring engineering cost.

In 1996 an ad hoc group of cryptographers and computer scientists claimed that a small business could produce an FPGA board that would break a key in 556 days, at a cost of \$10,000. As a consequence, they suggested that the DES key be expanded to 90 bits. Goldberg and Wagner disagreed with the ad hoc group and implemented DES in an Altera FLEX8000 part [4]. They extrapolated their results for the algorithm to project a cost of \$45,000 for a full cryptanalytic machine that would search through the entire key-space in one year. There is much published research on custom hardware cryptanalytic machines for DES, dating as far back as 1977 [6]. Wiener is currently considered the best estimate of custom hardware performance [7].

Gray and Kean implemented DES on the fine-grained Configurable Array Logic device. This design faithfully implemented the standard DES architecture, thus exploiting the bit-level parallelism but not partial evaluation.

Villasenor et al. used partial evaluation to accelerate an automatic target recognition system [8]. Singh, Hogg, and McAuley discuss the use of partial evaluation for the dynamic reconfiguration of FPGAs [9]. They made several observations about what would constitute a good application for partial evaluation of hardware in FPGAs. The first was that the relevant inputs should be pseudo-static, i.e., remain constant for a long period of time. Partially evaluated hardware will obviously bring about an increase in the number of configurations and, because configuration times can be long, designers should be sure that the circuit improvements justify incurring the time overhead. Additionally, the circuit changes resulting from the partial evaluation should be small, so as to minimize the configuration time. This fact calls for a partially reconfigurable FPGA device.

Finally, in the context of CAD support for dynamic reconfiguration, they made the point that an efficient method of creating/modifying the file that programs the FPGA is an important design challenge for partially evaluated hardware. Perhaps the most systematic use of partial evaluation and hardware circuits involves the PECompiler developed by Wang and Lewis, which attempts to automatically apply these techniques without relying on domain- or application-specific information [10]. Their work is in a very early stage, however.

This work makes two specific contributions. While a number of researchers have used partial-evaluation techniques, none have gone as far as the work here in removing subsequent control hardware. Thus, we believe that the DES design used here is the most complicated and advanced application of partial-evaluation to date. Secondly, this is the first known design of a cryptography system that capitalizes on key-specific hardware to achieve a cost or performance advantage.

3. DES Algorithm

DES is a 56-bit private key encryption algorithm based on a block cipher with a 64-bit block. Block ciphers deal with blocks of data as separate units, rather than operating on a continuous stream. The input is known as the plaintext while the encrypted output is called the ciphertext. The algorithm contains 16 rounds (iterations) of identical operations performed with a set of 48-bit subkeys. In typical ASIC implementations, a single circuit with registers and a feedback loop implements the iteration control and a second circuit generates each 48-bit subkey from the 56-bit key. The same datapath is used to perform both encryption and decryption. A single key is used for both encryption and decryption, but the subkeys are generated in the reverse order. Because of these properties, DES is called a symmetric private key cryptosystem. Encryption begins with an initial permutation (IP), which scrambles the 64-bit plaintext in a fixed pattern. The result of the initial permutation is sent to two 32-bit registers, called the right-half register and left-half register. These registers hold the two halves of the intermediate result through the succeeding 16 iterations.

The contents of the right-half register are permuted (Permutation E) and sent to an exclusive-OR unit along with the subkey for each iteration. Note that some bits are selected twice, allowing the 32-bit register to expand to 48 bits. The product of the exclusive-OR block is used to address a set of eight substitution memories (S-boxes). The inputs to S-box S1 are generated through the exclusive-OR of: bit 32 from the right-half register with bit 1 from the subkey, bit 1 of the right-half register and bit 2 of the subkey, etc. The subkey is directly applied to the exclusive-OR block of the S-box inputs, while some bits from the 32-bit right-half register must be selected twice to obtain the 48 bits of input for the eight S-boxes. The result of the exclusive-OR forms the inputs to eight 6-bit input to 4-bit output S-boxes. The S-box outputs are permuted (Permutation P) and fed into an exclusive-OR block along with the contents of the left-half register. The output of this block is written into a temporary register, concluding the first iteration. At the next clock cycle, the contents of the temporary register are written into the right-half register and the previous contents of the right-half register are written into the left-half register. This process repeats through 16 iterations. After the sixteenth iteration, the right-half and left-half register contents are subjected to a final permutation (IP^{-1}), which is the inverse of the initial permutation. The output of IP^{-1} is the 64-bit ciphertext.

The 56-bit key is subjected to a permutation (PC-1) and the result of the permutation is stored in two 28-bit registers, C and D. The 56-bit key is often transmitted as a 64-bit block with every eighth bit acting as a parity bit. At the beginning of each iteration the contents of these two registers are rotated by either one or two bit positions, based on the iteration count. Finally, 48-bits from the two registers undergo a permutation (PC-2) to generate the subkey for each iteration. The first 24 bits of the subkey come from the C register, while the last 24 are from the D register. Four bits from each of the registers are unused during each iteration.

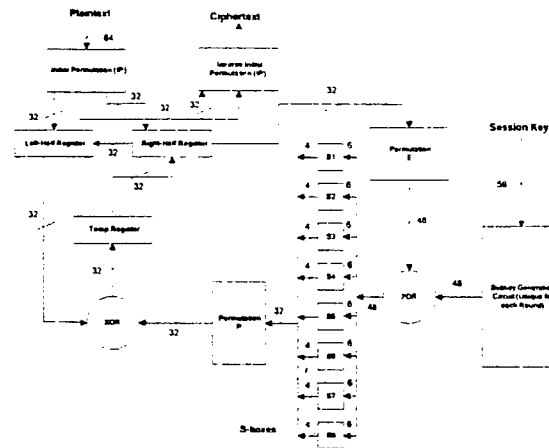


Figure 1. DES Algorithm

4. Partial Evaluation and DES

The key technique presented here for improving DES is to apply partial evaluation to the circuit design. Partial evaluation is a general-purpose optimization technique which involves transforming a function $F(A,B)$ into a new function $F_A(B)$. This new function is optimized under the assumption that some of the inputs to the function, namely the set A , are known. Such a transformation is useful if the designer can make some valuable optimizations as a result of knowing the value of A . DES has three inputs: the plaintext, the session key, and the encrypt/decrypt select line.

In many secure communications systems, the encryption key will remain constant during the complete communication session. The encryption key is sometimes referred to as a session key for this reason. In an ASIC or non-adaptive FPGA implementation, the key will be stored in a 56-bit register and subjected to the shifting and permutations to generate the 48-bit subkey for each of the 16 iterations of the algorithm. The subkey bits then become the inputs to the exclusive-OR block along with bits from the intermediate message halves.

However, because for most usage scenarios the session key is pseudo-static for a relatively long period of time, there exists an opportunity to specialize the DES circuitry. The possibility exists to generate the sixteen subkeys once and then select between them with a multiplexer. This basic approach is particularly well suited to

FPGAs because it removes several of the permutations and much of the logic and routing. These functions are trivial with ASIC technology, where routing is very flexible, but they are expensive for FPGAs.

To understand the nature of this specialization, note that an exclusive-OR gate with inputs A and B can be thought of as a 2:1 multiplexer controlled by B: it selects either input A or its inverse. In the case of DES, the inputs to the multiplexers are inverted and non-inverted versions of the intermediate message half bits. The control lines to the multiplexers are functions that reflect the subkey bits at specific iterations, i.e. hard-wired functions of the iteration count. By optimizing the circuit for a specific session key, we were able to remove the following resources:

- the 56-bit session key register
- the 48-bit subkey register
- all of the shifting circuitry
- permutation routing
- the 48 exclusive-OR gates

This hardware is replaced by 48 2:1 multiplexers that are controlled by the iteration counter. This optimization assumes that the encrypt/decrypt line is kept constant, i.e. the subkeys are for either encryption or decryption but not both. The design can encrypt and decrypt if the control lines to the multiplexers become functions of five bits. We will present results for both approaches.

To see the effects of the partial evaluation more clearly, consider the VHDL code for the two designs (Figure 2). We will refer to the full DES design, with all of the original circuitry for generating the sixteen subkeys, as the static design. The dynamic design has been specialized for a known session key. Recall that the output of the subkey generation circuitry leads to 48 exclusive-OR gates which are used to address the S-boxes. The static design has four processes for addressing the S-boxes. Two of the processes are used to implement permutations. The hardware in this portion of the static design is represented by the "ks" process (key scheduling), which performs the shifting function and the "lut_in" process (lookup table input), which implements the exclusive-OR block. By contrast, the dynamic design contains only the "lut_in" process for addressing the S-boxes. This process consists of a "case" statement, with the address bits conditionally inverted as a result of the intermediate message half bits. The selection of "Sb_inp(index) <= rh(bit)" or "Sb_inp(index) <= not rh(bit)" reflects the value of the subkey *index*-th bit. The subkey bits in turn reflect select session key bits, with selection dependent upon the iteration count.

The code shown is for a design that assumes either encrypt or decrypt but not both. For a design that would retain the option to encrypt and decrypt, an "if" statement would precede the "case" statement, followed by an "else" "case" statement, making the S-box inputs functions of five variables instead of four. Due to the width of the buses, the processes are all abbreviated in the figure.

5. Results

A static DES design and two dynamic designs with 32-bit interfaces were placed and routed into Xilinx XC4000 series parts. Synthesis was initially done with Synopsys and the XACT tools were used for place-and-route. The first dynamic design assumed that the session key and value of the encrypt/decrypt signal are

<pre> PERM1 : PROCESS (KEY) BEGIN SK_CPERM(1)<=KEY(57); SK_CPERM(2)<=KEY(49); ... END PROCESS; -- KS : PROCESS (C, ENCRYPT) -- SUBC & SUBD ARE 28-BIT VARS BEGIN SUBC := SK_CPERM; SUBD := SK_DPERM; IF (C'EVENT AND C='1') THEN IF ENCRYPT='1' THEN IF COUNT = "0000" THEN SKC<=SC(2 TO 28)&SC(1); SKD<=SD(2 TO 28)&SD(1); ELSIF COUNT = "0001" THEN SKC<=SC(3 TO 28)&SC(1 TO 2); SKD<=SD(3 TO 28)&SD(1 TO 2); ... END IF; ELSE IF COUNT="0000" THEN SKC<=SUBC; SKD<=SUBD; ELSIF COUNT="0001" THEN SKC<=SC(28)&SC(1 TO 27); SKD<=SD(28)&SD(1 TO 27); ... END IF; END IF; END IF; END PROCESS; -- PERM2 : PROCESS (SKC, SKD) BEGIN SK(1)<=SKC(14); SK(2)<=SKC(17); ... END PROCESS; -- LUT_IN : PROCESS (RH, SK) BEGIN SB1_IN(1)<=RH(32) XOR SK(1); SB1_IN(2)<=RH(1) XOR SK(2); ... SB8_IN(5)<=RH(32) XOR SK(47); SB8_IN(6)<=RH(1) XOR SK(48); END PROCESS; </pre>	<pre> LUT_IN : PROCESS (COUNT) BEGIN CASE COUNT IS WHEN "0001" => SB1_IN(1)<= RH(32); SB1_IN(2)<= RH(1); ... WHEN "0010" => SB1_IN(1)<= NOT RH(32); SB1_IN(2)<= RH(1); ... SB8_IN(5)<= RH(32); SB8_IN(6)<= NOT RH(1); ... WHEN OTHERS => NULL; END CASE; END PROCESS; </pre>
Static Design	Dynamic Design

Figure 2. Code Comparison for Static and Dynamic Design

constant, while the second merely assumed the session key was constant. Figure 3 shows the floorplan of the static design implemented in the XC4025 part. Subkey generation hardware, both the function generators and the registers for the 56-bit session key and the 48-bit subkey, occupy much of the right side of the device. By contrast, Figure 4 shows the floorplan for the first dynamic design implemented in the same XC4025 part. There are far fewer occupied configurable logic blocks (CLBs) than for the static design, thus decreasing the burden on routing and logic resources.

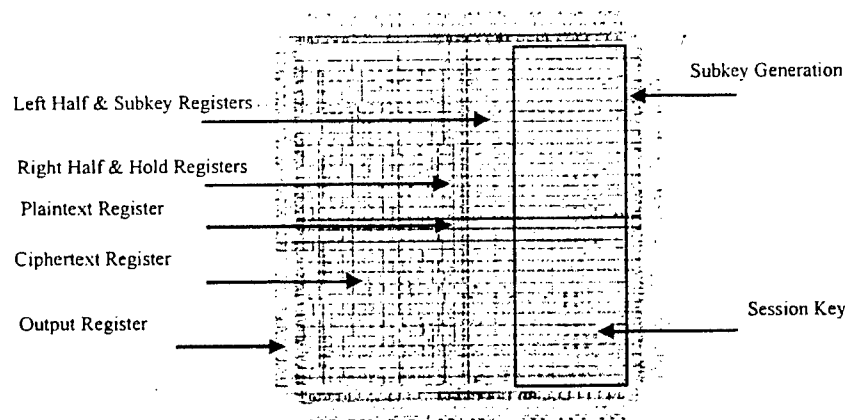


Figure 3: Floor plan of static design, with subkey scheduling circuits.

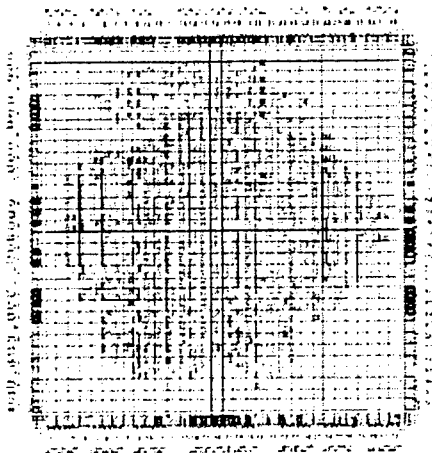


Figure 4. Floorplan of Dynamic (i. e. Partially Evaluated) Design

The resource and performance results are shown in Table 1. The static design required 938 CLBs, while the first dynamic design required 520 and the second 640. The static design was composed of 1103 FG function generators and 197 H function generators. The numbers were 702 FG function generators and 196 H function generators for the key and encrypt/decrypt specific design. The total number of

function generators for the first dynamic design was 69% of that for the static design, while the number of occupied CLBs was 55% of the static design. The total number of flip-flops was 365 for the static design and 261 for the first dynamic design.

By inspecting the static design, it was found that 480 FG function generators and 48 H function generators, plus 104 flip-flops for the key and subkeys, went to the subkey generation hardware alone. By removing this circuitry the critical path was shortened and routing constraints loosened. This factor allowed the placement tools greater flexibility, thus producing a more efficient route for the remaining circuits. By reducing the number of occupied CLBs, both of the dynamic designs could be placed in the 4013 while the static design required a 4025. Results from the second dynamic design are reported for both 4013-4 and 4025-4 parts (-4 indicates a speed grade for the device). Interestingly, the performance is significantly better in the larger 4025 part. This indicates that the routing requirements are an important factor for the smaller part. This phenomenon did not occur with the first dynamic design, which uses even fewer resources: performance was nearly identical for 4013-4 and 4025-4. The clock speed for the static design was 4.9 MHz, or 2.45 MB/sec in the 4025-4 (16 clocks per result, 8 bytes per result). The clock speed for the first dynamic design was 6.6 MHz in the 4013-4, which corresponds to a rate of 3.3MB/sec. This rate is a 45% speed improvement over the static design. The clock speed for the second dynamic design was 6 MHz, corresponding to a data rate of 3MB/sec. The speed numbers are based on the Xdelay program, which is conservative: implemented speeds are possibly higher. Also note that these numbers are based on a specific part and speed grade. To compare these numbers to ASIC and software implementations see the *Applied Cryptography* text by Bruce Schneier [1]. He lists VLSI Technology's 6868 chip as the highest performance ASIC at 64 MB/sec. The fastest listed software implementation is on an HP 9000/887 with a clock speed of 125 MHz, and a DES bandwidth of approximately 1.6 MB/sec.

	Full Static Design	Dynamic Without Decrypt	Dynamic With Decrypt	Dynamic Design With Decrypt
Part	4025-4	4013-4	4025-4	4013-4
Occupied CLBs	938	520	640	527
FG Funct Gen	1103	702	711	711
H Funct Gen	197	196	181	181
Flip-Flops	365	261	261	261
I/O Pins	75	73	74	74
Clock Speed	4.9 MHz	6.6 MHz	6MHz	5.2MHz
Data Rate	2.5 MB/sec	3.3 MB/sec	3 MB/sec	2.6 MB/sec

Table 1. Static vs. Dynamic Designs on Xilinx XC4000 Parts

To obtain a better idea of the difference in resource usage for the three designs, they were each synthesized with Synplify from Synplicity and targeted to the ORCA 2C family from AT&T. To understand how the results should be compared considering the change in synthesizers, the previous three designs were also synthesized with Synplify and targeted to the Xilinx 4000. The smaller dynamic design has about the same percentage of resources relative to the static design at 62%,

but the absolute number is much greater for all three designs. Synplify has fewer options than Synopsys and produces lower quality circuits (for these cases) in less time. The difference for the ORCA part are less than on the Xilinx part, but still significant. For instance, in the dynamic design with encrypt mode, the number of 4-input look-up tables is 80% of the number for the complete static design. This result indicates that the resource savings can be extended across FPGA architectures.

6. FPGA Technology Considerations

Key-specific hardware requires an efficient way of generating the FPGA configuration bitstreams for each of the 2^{56} possible session keys. In order for partial evaluation to be used in a deployed system, these bitstreams must reflect the effects of the changes in the VHDL code discussed in the last section without actually generating new code and running the backend tools. Obviously, the full set of bitstreams cannot be stored in memory; they must be generated in response to activating a specific session key. Fortunately, a large portion of the circuit remains the same for all keys. Keeping the same placement for those parts of the circuit reduces the changes in the bitstreams from one circuit to another. We are in the process of measuring the number of bits that must be modified as the session key changes. The next step in our research will be to generate a software tool that reprograms the affected bits directly as a function of a specific key.

While reducing the amount of circuitry needed to generate bitstreams for the desired keys is obviously important, there is another component of circuit creation that contributes to delay. In the Xilinx XC4000 parts and many other FPGAs, the entire configuration must be loaded into the device in order to cause even the smallest modification. This application, and in fact most configurable computing applications, benefits from low configuration time. In fact, configuration time will almost certainly determine whether or not the partial evaluation approach of DES is right for a given operating scenario. For instance, if this approach were used in a code-breaking machine the configuration time would have to be added to the processing time, as each key is cycled through once. In more typical communications applications, the frequency of key changes would determine the applicability of partial evaluation. Recalling that blocks are only 64 bits and considering the size of transmissions under a single key, the configuration time can actually be seen to be negligible in many cases. In those cases, the speed advantage of the partial evaluation design over the static FPGA design can be an important consideration.

Partial reconfiguration would be extremely useful for avoiding large configuration times when large portions of a circuit are truly static. In these cases, the whole configuration need not be loaded. Instead, just those portions that change are loaded. As stated earlier, the changes to the partially evaluated DES circuit are small. Section 4 showed the changes in the VHDL code necessary to go from key to key. A "case" statement invoked a function of four bits. A signal, the S-box input, was assigned one of two values depending on the key. In the circuit, such changes result in different configurations for function generators. Thus, partial evaluation of DES would greatly benefit from partial reconfiguration of the FPGA. The changes to the circuit are small enough to make it a good option, and reducing the configuration time makes the proposed way of implementing DES in an FPGA appropriate for a greater number of applications.

7. Conclusion

The advantages of partial evaluation for DES have been described. In the Xilinx XC4000 family FPGAs, one example of a dynamically generated design was shown to consume 69% of the function generators of a fully static DES design. Additionally, the dynamic design improved the speed of encryption from the 2.45 MB/sec of the static design to 3.3 MB/sec. To extend the results beyond the Xilinx part, resource utilization in the AT&T ORCA 2C part was also compared. The dynamic design consumed 80% of the 4-input look up tables that the static design consumed. The necessity for dynamic configuration generation was also discussed for this design.

Acknowledgements: This work was supported by the Defense Advanced Research Projects Agency of the United States of America, under contract DAB763-95-C-0102 and subcontract QS5200 from Sanders, a Lockheed Martin company.

References

- [1] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. New York: John Wiley & Sons, 1996.
- [2] J. P. Gray and T. A. Kean, "Configurable Hardware: A New Paradigm for Computation," presented at Decennial Cal Tech Conference on VLSI, Pasadena, CA, 1989.
- [3] M. Blaze, W. Diffie, R. L. Rivest, B. Schneier, T. Shimomura, E. Thompson, and M. Wiener, "Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security: A Report by an ad hoc Group of Cryptographers and Computer Scientists," <http://www.bsa.org/policy/encryption/cryptographers>, 1996.
- [4] I. Goldberg and D. Wagner, "Architectural Considerations for Cryptoanalytic Hardware," <http://www.cs.berkeley.edu/~iang/isaac/hardware>, 1996.
- [5] S. Son, "Feasibility Study: DES Cryptanalysis Using Reconfigurable Hardware," <ftp://ftp.et.byu.edu/papers/desfpga.ps>, 1996.
- [6] W. Diffie and M. E. Hellman, "Exhaustive Cryptanalysis of the NBS Data Encryption Standard," in *IEEE Computer*, vol. 10, 1977, pp. 74-84.
- [7] M. J. Wiener, "Efficient DES Key Search," presented at CRYPTO '93, Santa Barbara, CA, 1993.
- [8] J. Villasenor, B. Schoner, K. Chia, C. Zapata, H. J. Kim, C. Jones, S. Lansing, and W. H. Mangione-Smith "Configurable Computing Solutions for Automatic Target Recognition," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, J. Arnold and K. L. Pocek, Eds. Napa, CA, 1996, pp. 70-79.
- [9] S. Singh, J. Hogg, and D. McAuley, "Expressing Dynamic Reconfiguration by Partial Evaluation," presented at Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, 1996.
- [10] Q. Wang and D. M. Lewis, "Automated Field-Programmable Compute Accelerator Design Using Partial Evaluation," presented at Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, 1997.

FPGA FINGERPRINTING TECHNIQUES FOR PROTECTING INTELLECTUAL PROPERTY

John Lach¹, William H. Mangione-Smith¹, Miodrag Potkonjak²

The Departments of Electrical Engineering¹ and Computer Science² at the University of California
Los Angeles, California, USA

Abstract

As CAD tools and semiconductor technology improvements increase market opportunities for reusable hardware components, it becomes more important to produce techniques for protecting intellectual property rights. This work presents a method of fingerprinting an FPGA design component, so that products in the field can be used to identify both the component designer as well as the customer of record. These techniques are efficient, have extremely low impact on design quality, and are resistant to tampering.

Introduction

We will introduce a fingerprinting technique that applies cryptographically encoded signatures to FPGA digital designs in order to support identification of the original recipient (i.e. customer of record). The approach is shown to be capable of encoding long signatures and being secure against malicious collusion while being extremely efficient and requiring low overhead in terms of area and design performance.

A. Motivation

Digital IC design implementation has dramatically increased in complexity. Fortunately, complex systems tend to be assembled using smaller components in order to reduce complexity as well as to take advantage of localized data and control flows. This trend toward partitioning enables design reuse, which is essential to reducing development cost and risk while also shortening design time. Design reuse has been employed by systems designers for years, but what is new is that the boundaries for component partitions have moved inside of the IC packages.

These reusable modules are commonly referred to as intellectual property (IP), as they represent the commercial investment of the originating company but do not have a natural physical representation. Direct theft is a major concern of IP vendors. It is possible for customers, or a third party, to simply sell an IP block as their own without even reverse engineering the design. Because IP blocks are designed to be modular and integrated with other system components, the thief can simply repackage them without bothering to understand either the architecture or implementation.

This paper presents a solution to the risk of such direct misappropriation. The essential idea involves embedding a digital signature, which uniquely identifies the recipient, in an IP block. This signature allows the IP owner to not only verify the physical layout as their property but to identify the

source of misappropriation, in a way that is likely to be much more compelling than the existing option of verifying the design against a registered database. This capability is achieved with very low overhead and effort and is protected against recipient collusion.

B. Motivational Example

While the concepts developed here can be applied to a wide range of FPGA architectures, all of the discussion and experimental work will be conducted in the context of the Xilinx XC4000 architecture [1]. These devices are composed of an array of configurable logic blocks (CLBs), each of which contains two flip-flops and two 16x1 lookup tables (LUTs). A hierarchical and segmented routing network is used to connect CLBs in order to form a specific circuit configuration.

Using a previously developed FPGA watermarking technique [2], a secure transparent signature can be placed in an FPGA design. Consider the case of PREP Benchmark #4, a large state machine, which can be mapped into a block of 27 CLBs. This mapping results in 3 unused CLBs, or $3 \times 32 = 96$ unused LUT bits. Each unused LUT bit is used to encode one bit of the signature. Fig. 1 shows the layout of the original design as produced by the standard Xilinx backend tools, while Fig. 2 shows the layout for the same design after applying the watermark constraints to the three unused CLBs and re-mapping the design. The marked CLBs are then incorporated into the design with unused interconnect and neighboring CLB inputs, further hiding the signature.

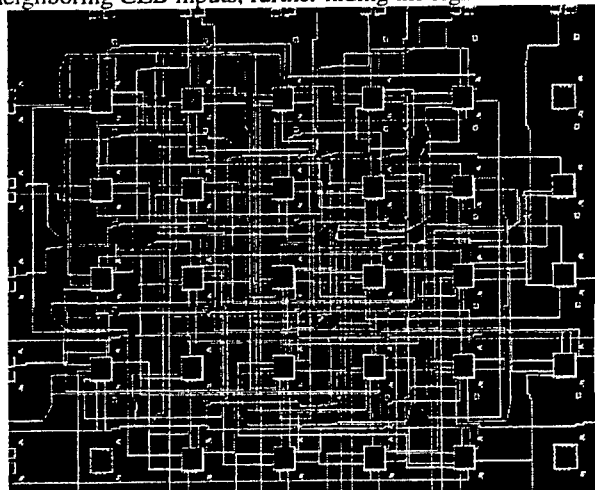


Fig. 1: Original layout of PREP benchmark #4

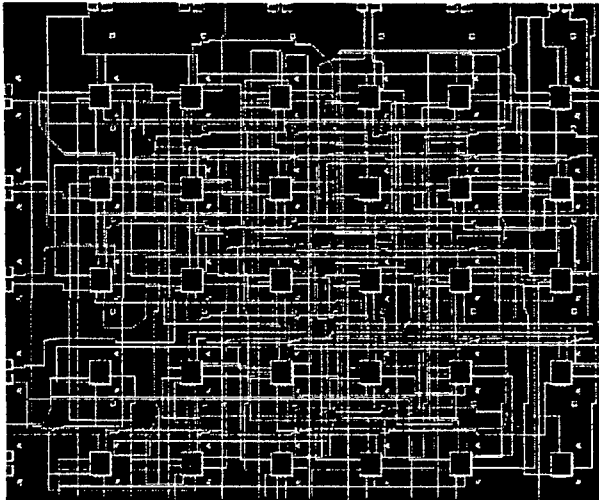


Fig. 2: Watermarked layout of PREP benchmark #4

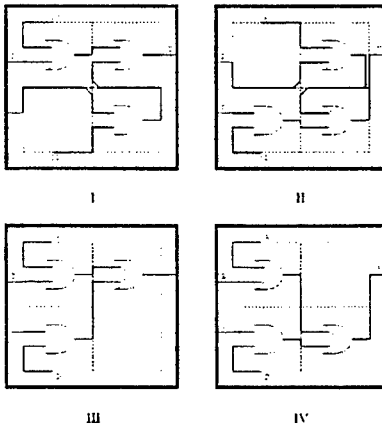


Fig. 3: Four instances of the same function with fixed interfaces

An FPGA design partitioning and tiling technique [3] is then used to extend the watermarking technique's capabilities to include fingerprinting. Consider the Boolean function $Y=(A \wedge B) \wedge (C \vee D)$, which might be implemented in a tile containing four CLBs as shown in Fig. 3. This configuration contains one spare CLB, making its LUT available for the insertion of a signature. Each recipient could receive this original configuration with a unique signature. Using the same configuration for a different recipient, and therefore a different signature, would facilitate simple comparison collusion (e.g. XOR), as the only difference between the designs would be the signatures. But, each implementation in Fig. 3 is interchangeable with the original, as the interface between the tile and the surrounding areas of the design is fixed and the tile's function remains unchanged. The timing of the circuit may vary, however, due to the changes in routing. With several different instances of the same design, comparison collusion would yield functional differences, thus disguising the differences between the various recipients' signatures.

C. Contributions

This paper presents the first fingerprinting method for protecting intellectual property, in the form of reusable digital circuits, even after the IP has been delivered in commercial products. By manipulating hardware resources, we are able to encode relatively long messages in a manner that is difficult to observe by a third party, resists tampering and collusion, and has little impact on circuit performance or size. This capability provides three main benefits:

1. It reduces the risk that a circuit will be stolen, i.e. used illegally without payment or transferred to a third party.
2. It can be used to identify the backend tool chain used to develop a design, and thus be part of the royalty mechanism used for CAD tools.
3. It identifies not only the origin of the design, but also the origin of the misappropriation.

Related Work

There has been a number of fingerprinting efforts reported in data hiding and cryptography literature [4-6]. They established a spectrum of protocols which guarantee the protection of both buyers and merchants of digital artifacts. All of them are targeting protection of still artifacts, such as image and audio streams. To the best of our knowledge, this is the first effort which addresses intellectual property protection using fingerprinting. This technique, therefore, provides protection of intellectual integrated circuit properties, such as cores, at a level beyond what is provided by recently introduced watermarking techniques [2, 7].

Approach

This fingerprinting approach makes use of both the watermarking technique and the design tiling technique to create secure signatures.

A. Watermarking

The watermarking technique is the general approach of inserting signatures in unused CLB LUTs as introduced in the motivational example above. Results from that project demonstrated that the area and timing overhead required for inserting large, cryptographic signatures in a design that identify the circuit origin is extremely low and is protected against most attacks. The only limitation of the approach is that a signature is fundamentally an optional component of a system design. Any signature can be removed by reverse engineering a design to a stage before the signature has been applied. For example, the approach developed here will be used to watermark a design at the physical level by manipulating LUTs and interconnect. The IP vendor will then deliver their technology in the form of a hard macro. If the macro can be reverse engineered to a netlist, the signature will be removed, specifically because it is not a functional part of the circuit operation. A thief can then move forward

through the place and route tools to derive a hard macro that does not contain the signature. Fortunately, most FPGA vendors have taken a business position that they will not reveal the specification of their configuration streams, specifically to complicate the task of reverse engineering and thus protect the investment of their customers. Xilinx does not take any specific actions to make their configurations difficult to reverse engineer. However, they do believe that it is difficult to do in general, and they promise their customers that they will keep the bitstream specification confidential in order to raise the bar for reverse engineering [8]. Essentially, once a design is watermarked, the only option for unauthorized removal is to reverse engineer back to a functional netlist and re-place and route the design.

Directly applying the watermarking technique to fingerprinting (i.e. replace the design origin signature with the recipient's signature for each copy) is susceptible to collusion. Performing a simple comparison between the two bitstreams would reveal that the only differences were due to the individual signatures. Removing the differences would yield a fully functional yet unmarked circuit.

B. Tiling

This problem can be avoided by taking advantage of the flexible nature of FPGAs to create functional differences between design instances. By moving the location of the signature for each instance of the design (i.e. reserve different CLBs for the signature), the functional design will also have a different layout. Therefore, all comparisons that are done yield functional differences, and any attempt to remove the differences would yield a useless circuit.

The design tiling algorithm was developed in connection with a fault-tolerance project, but it applies equally well here. The algorithm divides a design into a set of tiles that possess the same characteristics as the example in Fig. 3. That is, each tile has set specific functionality and locked interface with the rest of the design. Several instances of each tile can be generated, and each instance can replace another without affecting the rest of the circuit (except timing) due to the locked interface. For the fault-tolerance project, different instances of each tile reserved different CLBs as unused. In the face of a CLB fault, the appropriate instance can be activated without affecting the rest of the circuit. The same result could be achieved by storing several instances of the entire design, leaving various CLBs free in each instance, but the effort to place and route each instance and the memory required to store each instance makes this approach impractical.

Much in the same way that tiling for fault-tolerance reduces the effort required to generate the various fault-tolerant instances and the memory to store them, tiling also makes this fingerprinting approach more efficient and practical. Generating an entire layout for each instance of the design would require a trip through the place-and-route tools for the entire circuit. Tiling requires that only a small portion

of the design be changed, as the tiles are independent due to the locked interface between tiles. The various tile instances can then be matched to create one instance of the entire design. This reduces the total number of instances that can be generated, but vastly reduces the effort and memory required to produce each instance.

C. Fingerprinting

After each instance for each tile is generated, the instances are prepared for the signature. Every unused CLB in each instance is incorporated into the design with unused interconnect and neighboring CLB inputs, and timing statistics are generated for each instance. Depending on the timing specifications of the design, some instances may be discarded. The remaining instances are collected in a database. For example, MCNC benchmark c499 can be divided into 6 tiles, each with 8 instances, creating the possibility for $8^6 = 262,144$ different instances of the total design.

When a copy of the design is needed for distribution, an instance from each tile is selected from the database and the recipient's signature is inserted in the unused CLBs of each tile.

A group of people colluding to remove their signatures from their instances of the design may be able to find that they have instance matches among some of their tiles, thus allowing for tile comparison collusion, but it extremely unlikely that matches will be found among all or even a large portion of the tiles. Therefore, the colluding recipients may be able to remove a small portion of their signatures, but the vast majority of the signatures will remain intact. The key to this approach is efficiently introducing wide variation among the functional parts of the designs as well, so that collusion cannot be used to separate functional components from identifying markers.

The following pseudo-code summarizes the approach:

```

1. create initial non-fingerprinted design;
2. extract timing and area information;
3. while (!complete) {
4.     partition design into tiles;
5.     if (!(signature size && collusion protection)) break;
6.     for (i=1; i<=# of tiles; i++) {
7.         for (j=1; j<=# of tile instances; j++) {
8.             create tile instance(i,j);
9.             if (instance meets timing criteria) {
10.                 incorporate unused CLBs into design;
11.                 store instance;
12.             } } }
13. for (i=1; i<=# of recipients; i++) {
14.     prepare signature(i);
15.     select tile instances from database;
16.     insert signature in unused LUTs;
17. }
```


Lines 1 and 2 initialize the process by establishing the physical layout for the non-fingerprinted design, on which all area and timing overhead is based.

Lines 3-12 perform the tiling technique, creating a database of tile instances. The variables for this section are signature size, collusion protection (level of security based on presumed number of collaborators), and timing requirements. Signature size and collusion protection affect the tiling approach, while the timing requirements define the instance yield (i.e. individual tile instances are accepted contingent upon their meeting the timing requirements).

Lines 13-17 are executed for each distributed instance of the design. Line 14 derives the unique recipient signature with asymmetric fingerprinting techniques [5, 6].

Experimental Results

We conducted an evaluation of the proposed approach on nine MCNC designs.

Table 1 shows the cost (area) metrics of the designs before and after the application of the fingerprinting approach. A number of factors complicate the task of calculating the physical resource overhead. The place-and-route tools will indicate the number of CLB's that are used for a particular placement. However, these utilized CLB's rarely are packed into a minimal area. Unused CLB's introduce flexibility into the place-and-route step that may be essential for completion or good performance. For example, the initial c880 design possesses a concave region that contains 42 utilized CLB's but also 10 unutilized CLB's (19%). Therefore, we will report overhead in terms of the area used by the fingerprinted design minus the total area of the original design, including unused CLB's such as the 19% measure above. The average, median and worst-case area overheads were 5.4%, 5.3%, and 9.8% respectively. The size of the signature that can be encoded is dependent on this overhead. If a larger signature is desired, extra CLB's can be added thus increases overhead linearly with the size of the additional signature length.

Table 1: Variation of resources used among AFTB's for each tile

Design	Original # of CLB's	Final # of CLB's	Final - Original Original
9sym	46	49	.065
c499	94	96	.021
c880	110	115	.045
duke2	93	100	.075
rd84	27	28	.037
planet1	95	100	.053
styr	78	81	.038
s9234	195	206	.056
sand	82	90	.098

Timing overhead is shown in Fig. 4. For each design, the instance yield (i.e. number of tile instances that meet the timing specifications / total number of tile instances) is shown as the timing specifications (measured as percent increase over the original, non-fingerprinted design timing) grow more

lenient. The results reveal that a 20% increase in timing yields approximately 90% of total tile instances as acceptable. Relatively small changes in a circuit netlist or routing constraints can often result in a dramatically different placement and a corresponding change in speed. It appears that the impact of fingerprinting on performance is well below this characteristic variance.

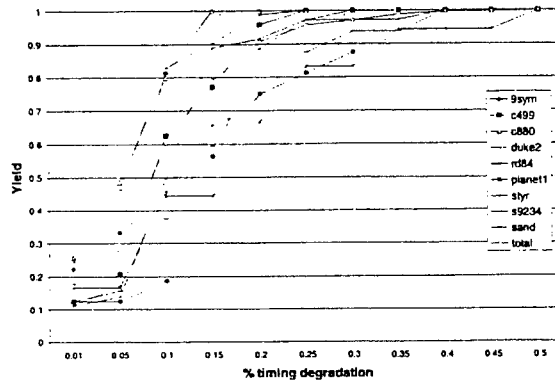


Fig. 4: Instance yield vs. timing specifications

Conclusion

As digital IC design complexity increases, forcing an increase in design reuse and third party macros distribution, intellectual property protection will continue to become more important. The fingerprinting approach presented here creates such protection for FPGA intellectual property, uniquely identifying both the origin and recipient of a design, while require very low overhead in terms of area and timing.

References

1. Xilinx, *The Programmable Logic Data Book*, 1996, San Jose, CA.
2. Lach, J., W.H. Mangione-Smith, and M. Potkonjak, "Signature hiding techniques for FPGA intellectual property protection", unpublished.
3. Lach, J., W.H. Mangione-Smith, and M. Potkonjak, "Low overhead fault-tolerant FPGA systems," *IEEE Transactions on VLSI Systems*, 1998 (special issue on FPGAs), in press.
4. Boneh, D. and J. Shaw, "Collusion-secure fingerprinting for digital data," *CRYPTO '95, 15th Annual International Cryptology Conference*, 1995, Berlin, Germany: Springer-Verlag.
5. Biehl, I. and B. Meyer, "Protocols for collusion-secure asymmetric fingerprinting," *STACS 97, 14th Annual Symposium on Theoretical Aspects of Computer Science*, 1997, Lubeck, Germany: Springer-Verlag.
6. Pfizmann, B. and M. Waidner, "Anonymous fingerprinting," *International Conference on the Theory and Application of Cryptographic Techniques*, 1997, Konstanz, Germany: Springer-Verlag.
7. Hong, I. and M. Potkonjak, "Behavioral synthesis techniques for intellectual property protection," unpublished, 1997.
8. Trimberger, S., personal communication, Xilinx, 1997.

LOW OVERHEAD FAULT-TOLERANT FPGA SYSTEMS

John Lach, William H. Mangione-Smith, Miodrag Potkonjak

jlach@icsl.ucla.edu, billms@icsl.ucla.edu, miodrag@cs.ucla.edu

56-125B Engineering IV

University of California

Los Angeles, CA 90095

Abstract - Fault-tolerance is an important system metric for many operating environments, from automotive to space exploration. The conventional technique for improving system reliability is through component replication, which usually comes at significant cost: increased design time, testing, power consumption, volume, and weight. We have developed a new fault-tolerance approach that capitalizes on the unique reconfiguration capabilities of FPGAs. The physical design is partitioned into a set of tiles. In response to a component failure, a functionally equivalent tile that does not rely on the faulty component replaces the affected tile. Unlike ASIC and microprocessor design methods, which result in fixed structures, this technique allows a single physical component to provide redundant backup for several types of components. Experimental results conducted on a subset of the MCNC benchmarks demonstrate a high level of reliability with low timing and hardware overhead.

1 INTRODUCTION

1.1 Motivation

While once FPGAs were mostly applied to prototyping, logic emulation systems and extremely low volume applications, they now are used in a number of high volume consumer devices. FPGAs are also now being used in more exotic applications. For example, the Mars Pathfinder mission launched in 1996 by NASA relies on Actel FPGAs for some system services. Unlike early applications, these high volume and mission critical systems tend to have stringent reliability requirements [1]. Thus, there is a drive from the user community to improve reliability through some level of fault-tolerance.

Unfortunately, current technology trends tend to make FPGAs less reliable. FPGA vendors have been moving down the same path of smaller device size as the rest of the semiconductor industry. Electronic current density in metal traces will increase as device feature size shrinks from 0.5 μm to 0.35 μm and smaller, which results in a greater threat of electromigration. As transistors shrink, the amount of charge required to turn them on reduces, which also makes the components more susceptible to gamma particle radiation. At the same time, FPGA vendors are moving to larger and larger dies in order to deliver more logic gates to their customers. The larger dies introduce more opportunities for failure and bigger targets for gamma particles.

Engineers traditionally respond to these threats through redundancy, such as replicating components (e.g. microprocessors and ASICs) or replicating logic internal to a component (e.g. Built-In Self-Repair (BISR)). However,

replication is a particularly unattractive approach for FPGA systems given the common customer complaint that devices cost too much and do not provide enough equivalent logic gates. A better approach is to leverage the flexible nature of FPGA devices to provide replication at a much finer level. Conceptually, if a single logic block fails, it is often possible to find an alternate circuit mapping that avoids the fault. Most vendor place and route tools provide an option for reserving resources, and in the face of a fault, the tool could be invoked to search for a new placement which only uses functional components. The resulting system could provide reliability with very low overhead, i.e. by reserving only a few percent of the resources as spares for fault recovery. Unfortunately, this approach results in significant system downtime. Thus, the technique will not be sufficient for mission-critical applications with hard real-time constraints. This approach also requires that the end user have the vendor place and route tools, which is usually not possible. It seems unlikely that the end consumer will wish to even know about an embedded FPGA, let alone worry about generating a new configuration for one. Finally, because each fault is distinct, each component would possibly require a unique circuit placement. These three factors combine to make the approach impractical.

We instead propose a technique for increasing FPGA system reliability with very low overhead. The target architecture for demonstration is a Xilinx 4000EX part, which is composed of an array of configurable logic blocks (CLBs). Nonetheless, we believe that this technique is applicable to a wide range of FPGA architectures. The place-and-route CAD tool maps a circuit net-list onto the array of CLBs and interconnect components. We propose partitioning the physical design into a set of tiles. Each tile is composed of a set of physical resources (i.e. CLBs and interconnect), an interface specification which denotes the connectivity to neighboring tiles, and a net-list. Reliability is achieved by providing multiple configurations of each tile. Furthermore, by using locked tile interfaces, the effects of swapping a tile configuration do not propagate to other tiles, thus reducing the storage overhead.

1.2 Motivational Example

Consider the Boolean function $Y=(A \wedge B) \wedge (C \vee D)$, which might be implemented in a tile containing four CLBs as shown in Figure 1. This configuration contains one spare CLB, which is available if a fault should be detected in one of the occupied CLBs. Upon detecting such a fault, an alternate configuration of the tile is activated which does not rely on the faulty CLB. Each implementation is interchangeable with the original, as the interface between the tile and the surrounding

areas of the design is fixed and the tile's function remains unchanged. The timing of the circuit may vary, however, due to the changes in routing.

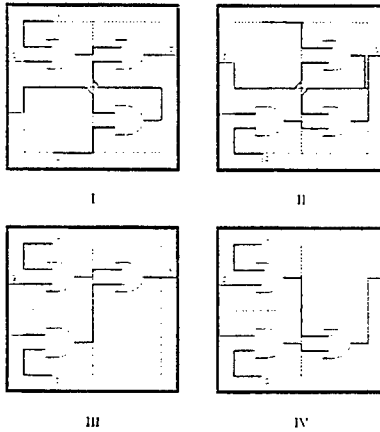


Figure 1: Motivational Example

This approach has three main benefits compared to redundancy-based fault-tolerance: very low overhead, the option for runtime management, and complete flexibility. The overhead required to implement this fine-grained approach, which can be measured in both physical resources on the FPGA (CLBs, I/O blocks, and routing) and timing, is extremely low compared to redundancy. Runtime management can be a very valuable feature of a system, particularly for mission-critical applications. This fault-tolerance approach handles runtime problems on-line, minimizing the amount of system downtime and eliminating the need for outside intervention. The flexibility that this approach provides allows for application specific solutions. The degree of fault-tolerance can be changed based on timing constraints, resource limitations, or presumed CLB reliability.

1.3 Paper Organization

The following two sections discuss background information, approach restrictions, and work related to fault-tolerance and FPGAs. Section 4 describes the details of the approach and implementation, and Section 5 introduces the formulas used to calculate the data for the experimental results in Section 6. Section 7 discusses future work on this topic, and the paper closes with some remarks summarizing the benefits of this fault-tolerance approach.

2 PRELIMINARIES

In this section, we survey the relevant background material for the proposed approach. We present the targeted FPGA architecture, the fault model assumed, and techniques envisioned for supporting the testing and fault diagnosis steps of the approach.

2.1 FPGA Architecture Model

The new fault-tolerance approach is demonstrated using the Xilinx XC4000EX family as the target architecture, specifically the XC4028EXBG352 [2]. However, neither the general concept nor the optimization algorithms are specific to

the 4000EX family, or even Xilinx architectures. Any FPGA architecture supporting the ability to reconfigure a large number of times could be used, such as the Altera 10k and the GateField flash memory devices. The approach is not applicable to anti-fuse systems, such as the Actel architecture, as they can not be reprogrammed.

2.2 Fault Model, Testing and Diagnosis

The proposed approach requires fault detection and a diagnosis method as a preprocessing step. We assume a widely used single stuck at, open, or short fault model [3]. It is interesting to note that our strategy actually covers many simultaneous faults, as long as each tile (see Sections 4 and 5) has at most one faulty CLB. In its current form, our approach does not address interconnect faults. Note that for local interconnects, interconnect faults will be expressed as a fault of the CLB to which it connects.

A number of schemes have been developed for detecting faults in FPGAs through exhaustive testing of the device architecture. Most of these approaches can be classified as off-line. For example, with Built-In Self-Test (BIST) [4-7], the FPGA is loaded with a small testing circuit that is restricted to a specific physical region of the device, which is then used to test another portion of the device. The test circuit is moved across the device in a systematic manner until the entire device is thoroughly tested. The downside of these approaches is that they require the device to be taken off-line, which may not be practical in highly fault-sensitive, mission-critical applications. Fault-detection latency also increases as a result of an off-line approach. Recently, an on-line testing scheme has been developed for bus-based FPGAs that avoids these problems [8].

3 RELATED WORK

Related work can be traced along the following three lines of research: FPGA synthesis, fault-tolerant design, and FPGA yield enhancement.

A number of different FPGA architectures and synthesis techniques have been proposed and demonstrated [9,10]. Conceptually, our fault-tolerance approach is closest to BISR techniques. The main targets for BISR are systems that are bit-, byte-, or digit- sliced. These types of systems include SRAM and DRAM memories [11], as well as systems designed using a set of bit planes and arithmetic-logic units (ALUs), assembled from ALU byte slices [1]. By far the most important use of bit-sliced BISR is in SRAM and DRAM circuits [12-14]. The bit-sliced BISR in memories significantly increases memory production profitability and is regularly used in essentially all modern DRAM designs. Among other BISR bit-sliced devices, the most popular and well addressed, from both a theoretical and practical point of view, are programmable logic arrays PLAs [15-18]. A simple, yet powerful methodology for the implementation of ALU byte slices was proposed by Levitt et al. [19].

Howard et al. [20] and Dutt et al. [21] have proposed using similar regularly structured BISR techniques for improving FPGA yield. Spare resources are allocated, and a

manufacturing step is used to swap spare CLBs for faulty components. Altera uses this approach, along with on-chip fuses, to increase production yield on the 10K parts. Mathur and Liu have proposed using modified place-and-route tools to reroute part of the net-list in the vicinity of a faulty CLB [22].

Our approach is completely transparent to the existing CAD tool chain and exists as an intermediate step that is used in conjunction with existing synthesis and place-and-route tools. Unlike the BISR techniques used in manufacturing, we are able to dynamically tolerate faults in the field. Finally, unlike Mathur and Liu, we are able to make timing guarantees (which is critical for real-time systems), require less system downtime, and do not require the end user to have access to FPGA CAD tools.

4 APPROACH

The key element of our approach to fault-tolerance is partially reconfiguring the FPGA to an alternate configuration in response to a fault. If the new configuration implements the same function as the original, while avoiding the faulty hardware block, the system can be restarted. The challenging step is to identify an alternate configuration efficiently. In this section, we elaborate on the key elements of our approach.

4.1 Tiles and Atomic Fault-Tolerant Blocks

We reduce the amount of configuration memory required by reducing the size of the component that is reconfigured. This is enabled by logically partitioning a design in a way that components can be independently reconfigured without impacting the rest of the design. In comparison with other alternatives, this approach also reduces the down time for devices that support partial reconfiguration and, more importantly, significantly increases the level of fault-tolerance with only nominal hardware and timing overhead. The key concepts for implementing the new approach are tiles and atomic fault-tolerant blocks (AFTBs).

Definition 1: A *tile* is composed of three elements: a set of CLBs and interconnect resources, a net-list which must be placed on those CLBs and routed across the interconnect, and a specification of how to interface the tile to adjacent tiles.

Definition 2: An *atomic fault-tolerant block* is one instance of a tile and has at least one spare CLB that serves to "cover" the faulty CLB(s).

Because each tile is associated with both physical resources and portions of the complete net-list, the design can only be partitioned into tiles after the complete net-list has gone through place-and-route once. By fixing the interfaces between the tiles, we create the opportunity to produce multiple partial configurations that satisfy the functional specification for a given tile, independently from the remainder of the design. Fault-tolerance is achieved by introducing spare resources into each AFTB so that, once a fault in a particular CLB is detected, a configuration of the tile's functionality that does not utilize the faulty CLB can be activated.

Each tile has a set of AFTBs. An AFTB is independent from all AFTBs associated with other tiles by virtue of the

fixed tile interface. Thus, selecting one AFTB for each tile can assemble a complete configuration, under the condition that none of the AFTBs rely on a faulty component.

Tiling provides many advantages in the implementation of fault-tolerant FPGA systems. First, the amount of memory needed to store the set of AFTBs is smaller than the amount required to store a set of complete configurations. For example, consider a design that must be able to tolerate any single CLB fault and which maps into a 6x6 CLB array. It may be possible to divide the design into four 3x3 tiles (Figure 2). Assuming that one configuration of the complete 6x6 design requires X bytes of memory, the non-tiled approach would require $36 \cdot X$ of memory for fault-tolerance: one configuration for each CLB that is at risk. With our method, each tile would require 9 AFTBs. However, since each tile ($X/4$ storage bytes) is independent, the entire storage is only $9 \cdot X$, a 75% reduction from the non-tiled approach.

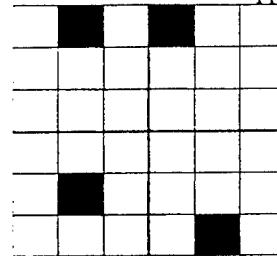


Figure 2: A 6x6 CLB design partitioned into 4 3x3 tiles

Tiling also increases reliability. For this example, the non-tiled approach could tolerate only one faulty CLB in the entire device. The tiled approach, however, is capable of tolerating any single fault in a tile but up to 4 faulty CLBs in the entire device.

The cost of increased fault-tolerance and reduced configuration memory is the possible introduction of more spare resources. For this example, the non-tiled approach reserves 2.7% of the CLBs to protect against a single fault, while the tiled approach reserves 11%. However, tiling opens up the opportunity to explore a rich design space. By choosing the tile size and amount of spare resources that are appropriate for system requirements, tiling provides a powerful tool to the designer.

One remaining issue involves circuit timing. While timing analysis tools are not completely reliable for FPGA devices, the Xilinx XACT Step software has proven to be reasonably accurate. We use this tool to determine the timing estimate of the initial configuration before tiling. Furthermore, because each individual AFTB is generated as a modification to the original configuration we have good timing estimates for any single failure. However, it is difficult to know what the circuit timing will be once multiple AFTBs have been activated in response to failures in more than one tile. In particular, the critical path for the entire FPGA could pass through several AFTBs without falling on the longest path within any of them.

4.2 Synthesis Methods

The synthesis approach is organized in an iterative top-down manner. We start with a non-fault-tolerant base design

and recursively partition it into tiles and AFTBs. We next check the feasibility of all fault scenarios in decreasing estimated level of difficulty. The idea is to terminate as early as possible those base designs that will not result in a feasible solution. We also calculate early the final reliability figures, so that the designer has an option of terminating the current search and starting one that has a higher reliability potential.

The synthesis is summarized in the following pseudo-code:

```

1. while (!(complete || design possibilities exhausted)) {
2.   create initial non_ft_design;
3.   extract timing and area information;
4.   calculate design reliability;
5.   while (!(complete || tiling possibilities exhausted)) {
6.     partition design into tiles;
7.     if (!meet area criteria) break;
8.     while (!(complete || AFTB possibilities exhausted)) {
9.       partition tiles into AFTBs;
10.      calculate AFTB reliability;
11.      if (!meet reliability criteria) break;
12.      order tiles by ft realization difficulty;
13.      order AFTBs by ft realization difficulty;
14.      for (j=1; j<=# of tiles; decreasing difficulty) {
15.        for (i=1; i<=# of AFTBs; decreasing difficulty) {
16.          create ft_design(i,j);
17.          if (!(success && meet timing criteria)) break;
18.        } } } } }

```

Lines 2-4 initialize the synthesis process for one instance of the base design. The place and route tool creates the base non-fault-tolerant design, and the relevant design characteristics are recorded. The procedure for the calculation of reliability is explained in Section 5.

Line 5 starts the synthesis algorithm and dictates that the loop will terminate upon the creation of a fault-tolerant design that meets all of the user specifications, including overhead (area and timing), level of fault-tolerance, and available memory. The loop will also terminate if the algorithm reaches the end of its exhaustive tile partitioning search, thus revealing that the specifications cannot be met for the given FPGA architecture and design generated in line 2. In this case, the complete algorithm will be repeated using a different base design with increased spare resources throughout the FPGA.

Line 6 partitions the design into tiles, as described in Section 4.1. The placement and shape of the tiles are determined by the following three key factors listed in decreasing order of importance: amount of interconnect across the tile interface, tile logic density, and tile size. Our reliability calculation (see Section 5) indicates that large tiles result in higher reliability. If the tiling attempt does not meet the user area specifications, the algorithm returns to the beginning of the tile partitioning loop for another tiling attempt. Hard macros (and fast carry chains) also affect the placement and shape of tiles, as efforts are made to keep macros intact.

Line 8 begins the AFTB partitioning algorithm, which terminates upon the successful creation of a fault-tolerant

design meeting all user specifications or upon the exhaustion of all AFTB partitioning possibilities. If the latter occurs, the algorithm returns to line 5 for re-tiling.

Line 9 lays out the various AFTBs within a tile. The number of AFTBs for a given tile depends on the desired level of fault-tolerance, the number of free CLBs in the tile, and the malleability and density of the logic. The criteria used for partitioning the design into tiles are also used for this AFTB partitioning.

Line 10 insures that the tile and AFTB partitions meet the user reliability specifications, and line 11 returns the algorithm to the beginning of the AFTB partitioning loop if they are not met. If upon such a return the AFTB partitioning possibilities have been exhausted, the design must be re-tiled, returning the algorithm to line 5.

Lines 12 and 13 facilitate early synthesis process failure detection. Tiles and AFTBs that are less likely to successfully place and route should be attempted first, thus efficiently returning the algorithm to the beginning of the loop if a fault-tolerant design meeting user specifications is not possible with the current tile and/or AFTB partitioning. The tile and AFTB characteristics causing them to be more difficult to realize include the criteria used in tile and AFTB partitioning. The presence of macros, and therefore reduced logic malleability, may also impact the assigned order of realization difficulty.

Lines 14 and 15 enforce the order defined by the two previous steps, as line 16 attempts to configure the various tiles and AFTBs. If the design can not be configured or if the configuration doesn't meet user timing specifications, the algorithm returns to the beginning of the AFTB partitioning loop (line 8) or, if AFTB partitioning is exhausted, to the beginning of the entire synthesis algorithm for re-tiling (line 5). The next iteration of line 6 partitions the design so as to give more slack (i.e. free CLBs) to the area of the previous iteration's failing tile. If no other tiling possibilities exist, the algorithm must return to line 1 for the creation of a new base design. If no base design possibilities remain, the algorithm terminates as unsuccessful.

The synthesis approach is illustrated using the PREP 5 benchmark shown in Figures 3-5. Figure 3 shows an implementation of the PREP 5 benchmark on the Xilinx 4000 architecture, a configuration that occupies rows 18-24 and columns 1-4. Only the CLBs in the defined area are used in the tiled design; the remaining CLBs are prohibited from use in any of the AFTBs.

Tile A covers rows 18-24 and columns 1-2, and tile B covers rows 18-24 and columns 3-4. Tiling was restricted by the occurrence of hard macros in each tile.

After partitioning the design into a set of tiles, the tiles are ordered by their implementation difficulty. In Figure 3, the tiles are ordered A first and B second, primarily because the logic in tile A is denser.

Next we create a set of AFTBs for each tile, which are sorted in decreasing order of implementation difficulty. The tiles in Figure 3 are assigned AFTBs that are composed of two adjacent CLBs. Each CLB is covered only once, making the total number of AFTBs per tile seven. For each AFTB, a

complete configuration, i.e. one AFTB for each tile, is passed through the place and route tool. Although the placement and routing of the logic in the tile can be quite different for each AFTB, variations within a tile do not propagate to other tiles (other than timing). This is possible because, for each fault-tolerant configuration, only the tile in question is changed. All other tiles remain the same as in the original configuration.

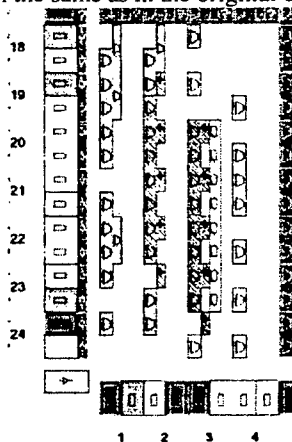


Figure 3: Initial floorplan for PREP 5 benchmark

Figure 4 shows the AFTB that was attempted first for the design in Figure 3.

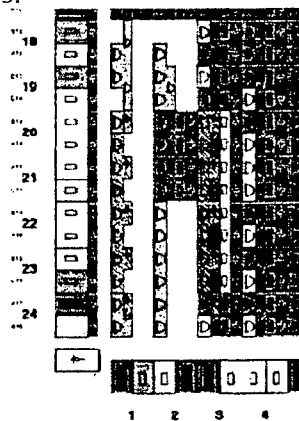


Figure 4: PREP 5 after tiling with one AFTB identified

4.3 Enforcing Fault-Tolerance at Run-time

After all of the AFTBs are stored in memory and the circuit begins operation, the system runs normally with the original configuration until a fault is detected. Upon detection, the circuit ceases functional mode until the proper reconfiguration can be made. As already mentioned, we assume that the fault detection system is able to identify the faulty resource in an architecture map. This information allows the system to retrieve the appropriate AFTBs from the configuration memory. The time needed for this memory access depends on normal access factors: access size, memory bus width, memory size, etc.

The last step involves the actual reconfiguration of the FPGA device. Once the AFTB is retrieved from memory, two options are possible depending on the capabilities of the FPGA architecture. If the device supports partial reconfiguration, the

AFTBs of the affected tiles can be used in isolation to directly update the configuration. Otherwise, the AFTBs must be merged with the active and functioning AFTBs, thus providing the necessary data for a total chip reconfiguration. For example, if the CLB at row 20, column 3 failed in the design of Figure 3, the proper configuration for tile B would be fetched from memory and configured onto the FPGA. The result is shown in Figure 5. Note that the interface between tiles A and B is unchanged.

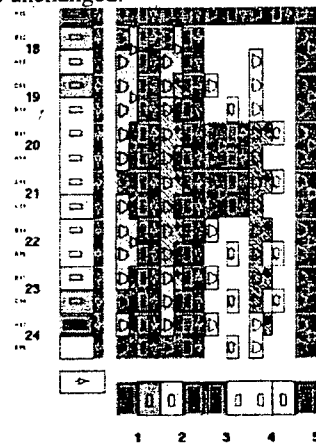


Figure 5: System at runtime after swapping the AFTB in tile B due to fault at (20,3)

The time required to update a tile with a new AFTB depends on the complete set of tiles, the device architecture and the surrounding computer system. However, in all cases, it is bounded and can be used to provide a measured level of system availability. After updating the affected tile, the device is reset and the system resumes operation as before the fault. The only possible change could be in the timing of the circuit, as the routing in the altered tiles has likely changed. Timing numbers are generated with each AFTB, and the system can operate under worst case assumptions. Another, technically more demanding, option is the use of a programmable clock. If new faults occur later, the process repeats itself until more than one fault occurs in a tile for which there is no available AFTB that has all faulty CLBs as unused.

5 RELIABILITY CALCULATION

The first step in calculating reliability is the selection of fault models. There are two major sources of logic faults in FPGA systems: cosmic radiation and manufacturing/operating imperfections.

Since the size of radiation particles is usually small when compared to the size of modern FPGA CLBs, we selected a cosmic radiation fault model that follows uniform distribution of independent (non-correlated) failures. Extensive terrestrial efforts to accurately model the rate of such soft faults indicate high variance (several orders of magnitude) depending on factors such as seasonal solar activity, altitude, latitude, device technology, and device materials. Even for the same chip from the same manufacturer, variations by a factor higher than 200 are not uncommon [23]. Experiments indicate that in FPGA-like devices at an altitude of 20 km, error rates significantly

higher than once per 1000 hours are common [24]. Also, as circuit devices become smaller, they become more sensitive to soft faults [23]. If one considers the multiyear life of computing devices and other sources of potential errors (e.g. power surges), the need for fault tolerance in devices which implement critical functions becomes apparent.

The second class of faults is related to manufacturing imperfections. These defects are not large enough to impact initial testing, but after a longer period of operation they become exposed. Design errors can also cause a device to stop functioning in response to rare sequences of inputs (e.g. due to a power density surge in a small part of design). For this type of model, we follow the gamma-distribution Stapper fault model [25]. The model is applicable to any integrated circuit with regular repetitive structure, including memories and FPGA devices.

In the remainder of the section, we elaborate on technical details related to the two fault models' reliability calculations.

5.1 Independent Uniformly Distributed Faults

Suppose that a design is partitioned into t tiles. Furthermore, assume that tile i has a total of c_i AFTBs. The total number of used CLBs in the initial design is t_n . For the sake of clarity and simplicity, we limit our discussion to the case where each AFTB consists of three or fewer CLBs. We denote by m_{1i} , m_{2i} , m_{3i} the number of AFTBs of size one, two, and three CLBs respectively in tile i . We also denote their weighted sum $m_i = m_{1i} + 2 * m_{2i} + 3 * m_{3i}$.

Finally, we assume that the probability of a CLB being faulty is $(1 - P)$, i.e. the probability that a CLB is fault free is P . It is easy to see that the probability, P_{init} , that the original design is fault free is $P_{init} = P^m$.

It is also easy to verify that the probability that a tile i in the optimized design is fault free, Pft_i , is given by the following formula:

$$Pft_i = P^{m_i} + m_i * P^{(m_i-1)}(1 - P) + (m_{2i} + 3 * m_{3i}) * P^{(m_i-2)} * (1 - P)^2 + m_{3i} P^{(m_i-3)} * (1 - P)^3$$

The first term corresponds to the scenario where all CLBs are fault free. The last three terms correspond to the scenarios where one, two, and three CLBs are faulty, respectively. The probability (Pft) that the optimized fault-tolerant design is

$$\text{functional is } Pft = \prod_{i=1}^t Pft_i.$$

5.2 Stapper's Fault Model

To calculate reliability for correlated faults, we started from the following formula [25]:

$$\bar{Y}_{mn} = \binom{n}{m} \bar{Y}_1^m \left(\prod_{i=0}^{m-1} \frac{\mu + i}{\mu + i \bar{Y}_1} \right) \times (1 - \bar{Y}_1)^{n-m} \left(\prod_{j=0}^{n-m-1} \frac{\mu + j \bar{Y}_1}{\mu + m \bar{Y}_1 + j \bar{Y}_1} \right)$$

(Note that we have fixed a small typographical error in the original published formula.)

The formula calculates the probability that exactly m out of n identical modules operate correctly for a given value of the variability parameter μ and single CLB reliability Y_1 . The parameter μ indicates the assumed or the measured probability of clustered faults. Small values of μ imply high levels of clustering. As μ tends toward infinity the formula reduces to the case of independent uniformly distributed faults.

Stapper's formula is used to calculate the probability that at least m out of n modules operate correctly by a direct summation of relevant terms.

6 EXPERIMENTAL RESULTS

We conducted an evaluation of the proposed approach and optimization algorithms in two phases. In the first, we applied the approach to nine MCNC designs. In the second phase we studied expected reliability improvement trends as a function of the number of used CLBs in a design.

Tables 1 and 2 show timing and cost (area) metrics respectively of the designs before and after the application of the new approach for reliability enhancement. The first column in both tables indicates the name of a design. The next four columns in Table 1 show the initial delay, and the best, worst, and median delay of the optimized designs. The rightmost column indicates the timing overhead as a result of enhanced reliability. For all nine designs, the largest timing overhead was in the range of 14% to 45%.

A number of factors complicate the task of calculating the physical resource overhead. The place-and-route tools will indicate the number of CLBs that are used for a particular placement. However, these utilized CLBs rarely are packed into a minimal area. Unused CLBs introduce flexibility into the place-and-route step that may be essential for completion or good performance. For example, the initial c880 design possesses a concave region that contains 42 utilized CLBs but also 10 unutilized CLBs (19%). Therefore, we will report overhead in terms of the area used by the fault-tolerant design minus the total area of the original design, including unused CLBs such as the 19% measure above. The area overhead is presented in Table 2, using the same format as Table 1. The average, median and worst-case area overheads were 5.4%, 5.3%, and 9.8% respectively.

Table 3 shows reliability improvements for the MCNC benchmarks under the uniform random fault model. The first column indicates the assumed probability (p) that a CLB is fault free. The next two columns show the probability that the original and fault-tolerant design of a particular benchmark is functioning properly. For example, for 9sym, with $p = 0.995$, the probability of the initial design and tiled design being functional is 81.0% and 98.4% respectively.

Table 4 shows the reliability figures for the same set of designs (original and tiled) with four different variability factors, μ , assuming the probability that a CLB is fault free is 90% and 99%. Table 5 is, in a sense, the strongest indication

Design	Initial (ns)	Fastest (ns)	Slowest (ns)	Median (ns)	Slowest - Fastest Fastest
9sym	71.6	71.6	82.0	76.8	0.15
c499	104.9	104.9	130.0	113.6	0.24
c880	110.8	110.8	126.4	117.3	0.14
duke2	87.9	87.9	118.8	96.4	0.35
rd84	50.2	50.2	72.8	58.6	0.45
planet1	145.0	145.0	194.9	166.1	0.34
styr	150.6	150.6	189.8	167.2	0.26
s9234	135.0	135.0	183.6	153.2	0.36
sand	97.6	97.6	117.7	103.8	0.21

Table 1: Timing bounds due to routing variation among AFTBs for each tile

Design	Original # of CLBs	Final # of CLBs	Final - Original Original
9sym	46	49	.065
c499	94	96	.021
c880	110	115	.045
duke2	93	100	.075
rd84	27	28	.037
planet1	95	100	.053
styr	78	81	.038
s9234	195	206	.056
sand	82	90	.098

Table 2: Variation of resources used among AFTBs for each tile

CLB P	.900		.950		.990		.995		.998		.999		.9999	
Design	Orig.	Tiled	Orig.	Tiled	Orig.	Tiled	Orig.	Tiled	Orig.	Tiled	Orig.	Tiled	Orig.	Tiled
9sym	1.2	16.9	11.6	56.2	65.6	95.7	81.0	98.4	91.9	99.5	95.9	100	99.2	100
c499	0.01	1.63	3.2	37.5	43.0	89.0	65.6	95.6	84.5	98.6	91.0	99.3	99.2	100
c880	0.0	0.6	1.8	31.7	37.3	91.2	61.2	97.6	82.2	99.6	90.7	99.9	99.0	100
duke2	0.01	0.7	2.8	31.9	41.3	90.8	64.3	97.5	83.8	99.6	91.6	99.9	99.1	100
rd84	7.2	38.6	27.7	74.7	77.8	98.5	88.2	99.6	95.1	99.9	97.5	100	99.8	100
planet1	0.02	5.4	11.5	32.6	41.7	94.1	64.7	98.4	84.0	99.7	91.7	99.9	99.1	100
styr	0.01	2.9	2.5	31.4	48.5	93.8	69.7	98.3	86.6	99.7	93.0	99.9	99.2	100
s9234	0.0	0.003	0.01	3.17	16.1	82.0	40.2	94.8	69.5	99.1	83.3	99.8	98.2	100
sand	0.03	1.53	1.83	2.50	45.7	92.4	67.6	97.9	85.5	99.7	92.5	99.9	99.2	100

Table 3: Reliability of the original vs. tiled designs against CLB reliability

	1		2		5		20	
	Orig.	Tiled	Orig.	Tiled	Orig.	Tiled	Orig.	Tiled
9sym	62.6/95.8	72.0/97.2	48.5/93.5	62.8/96.4	28.5/89.0	40.1/94.9	8.77/79.6	29.7/94.1
c499	57.9/95.1	68.2/96.5	41.6/92.2	54.8/95.5	19.9/86.1	36.2/94.1	2.76/71.6	14.5/88.7
c880	55.9/94.9	65.7/96.3	40.2/91.9	53.2/95.2	18.3/85.4	33.8/93.3	2.08/69.8	11.2/87.2
duke2	57.6/95.0	67.9/96.3	41.1/92.1	54.2/95.4	19.4/85.9	25.9/93.9	2.54/71.0	14.3/88.4
rd84	66.4/96.3	76.7/97.7	54.3/94.5	70.1/96.3	36.9/91.1	56.4/95.6	18.0/85.1	52.8/95.1
planet1	57.7/95.0	67.9/96.3	41.3/92.2	54.2/95.4	19.5/86.0	25.9/93.9	2.59/71.2	14.3/88.4
styr	58.9/95.2	68.1/96.9	43.0/92.5	56.2/95.8	21.6/86.8	39.6/94.4	3.63/73.4	15.4/89.8
s9234	53.1/94.3	64.4/95.6	35.1/90.9	58.9/93.5	13.1/82.9	28.4/89.6	0.63/62.5	5.32/83.6
sand	58.4/95.1	68.0/96.6	42.3/92.3	55.3/95.6	20.7/86.4	32.1/94.2	3.15/72.3	14.8/89.2

Table 4: Reliability of original and tiled designs using Stapper's correlated failure model with CLB reliability of 90%/99%

	CLB Overhead for Tiling	Random Fault Model		Stapper's Fault Model	
		Orig.	Tiled	Orig.	Tiled
9sym	6.5%	2.4	16.9	16.8	29.7
c499	2.1%	0.02	1.6	5.4	14.5
c880	4.5%	0.00	0.6	4.1	11.2
duke2	7.5%	0.02	0.7	5.0	14.3
rd84	3.7%	13.9	38.6	32.8	52.8
planet1	5.3%	0.04	5.4	5.1	14.3
styr	3.8%	0.02	2.9	7.1	15.4
s9234	5.6%	0.00	0.003	1.3	5.32
sand	9.8%	0.06	1.5	6.2	14.8

Table 5: Comparison of reliability and overhead for the original design with complete redundancy (i.e. 100% overhead) vs. tiled design for CLB reliability of 90% and $\mu = 20$

CLB	100 CLB design		1000 CLB design		5000 CLB design	
Reliability	Orig.	Tiled	Orig.	Tiled	Orig.	Tiled
.9500	0.005921	0.444669	0.000000	0.000302	0.000000	0.000000
.9750	0.079551	0.800119	0.000000	0.107534	0.000000	0.000014
.9800	0.132687	0.864375	0.000000	0.232820	0.000000	0.000684
.9850	0.220739	0.919633	0.000000	0.432660	0.000000	0.015161
.9900	0.366277	0.962643	0.000043	0.683364	0.000000	0.149026
.9950	0.606224	0.990317	0.006704	0.907280	0.000000	0.614762
.9980	0.819220	0.998429	0.136145	0.984404	0.000047	0.924414
.9990	0.905528	0.999608	0.370696	0.996091	0.007000	0.980610
.9995	0.951999	0.999903	0.611453	0.999028	0.085470	0.995153
.9999	0.990868	0.999995	0.912346	0.999952	0.632119	0.999762

Table 6: Reliability of traditional design methods vs. tiled approach against CLB reliability for large FPGAs

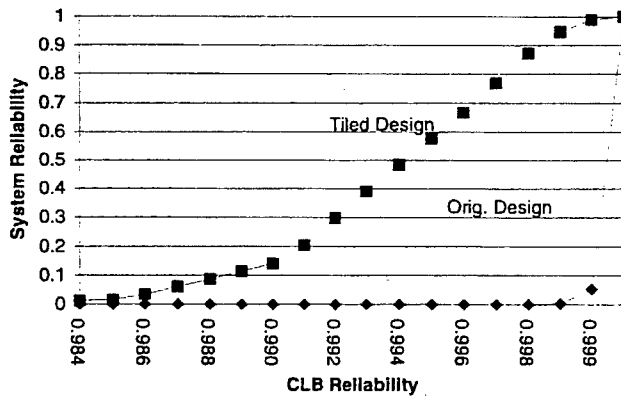


Figure 6: Reliability of traditional methods vs. tiled methods for a hypothetical 5000 CLB FPGA

of the effectiveness of the proposed approach for reliability improvement. The second column of this table indicates the area overhead of tiled designs. The next four columns provide reliability data under two selected fault models for the duplication-enhanced fault-tolerant original and for the tiled designs. For both models, the tiled designs have significantly lower area overhead and always higher reliability than the conventional fault-tolerant designs.

Finally, Table 6 calculates reliability improvement trends as the size of designs increase. It is assumed that all designs are partitioned into tiles of 5 AFTBs each consisting of two

CLBs and requires an average hardware overhead (i.e. ~5.4%). Table 6 indicates the potential of the proposed approach for reliability enhancement. For example, in the case of a 5000 CLB design, with $p = 0.999$, the probability of the initial design being functional is less than 1%, while the probability of the tiled design being functional is 98%. Figure 6 graphs the reliability results for the 5000 CLB design.

7 FUTURE WORK

Many of the highest volume FPGA devices tend to be dominated by interconnect resources, e.g. the Xilinx 4000 and

the Altera 10K families. On the Xilinx 4000EX series, the majority of configuration bits are used to program the state of the interconnect rather than the CLBs, and it is likely that these interconnect resources are more susceptible to faults. The fault-tolerance methodology presented above addresses faults in interconnect resources directly dedicated to specific CLBs because they appear as CLB faults. Unfortunately, the vast majority of interconnect resources pass through higher-level hierarchical switch structures that are not covered by unique CLB faults. Some of these routing resources will remain unused in each AFTB, thus providing some additional fault-tolerance. However, since this benefit comes as a byproduct of the approach rather than as a primary goal, we currently cannot make any specific claims on interconnect fault tolerance.

8 CONCLUSIONS

Fault-tolerant techniques have recently emerged as an important design consideration for FPGA-based systems due to the rapid progress in FPGA integration and the growing market for these devices. In order to address this problem, we have developed the first fault-tolerance approach to work at the level of physical design. Our hierarchical fault-tolerance technique partitions designs into tiles and atomic fault-tolerant blocks. The approach scales systematically through an exploration of the design solution space at the physical level. The approach is constructed of four phases: design partitioning, tile partitioning and ordering, AFTB partitioning and ordering, and reliability calculation.

Experimental results conducted on a subset of the MCNC benchmarks for large CLB FPGAs indicate that the technique is effective with low hardware overhead.

Acknowledgments

The authors would like to thank Prof. Jason Cong, John Peck, Hea Joung Kim, and Jason Leonard for their assistance. This work was supported by the Defense Advanced Research Projects Agency of the United States of America, under contract DAB763-95-C-0102 and subcontract QS5200 from Sanders, a Lockheed Martin company.


References

- [1] D. P. Siewiorek and R. S. Swartz, *Reliable Computer Systems: Design and Evaluation*, Burlington, MA, Digital Press, 1992.
- [2] Xilinx, *The Programmable Logic Data Book*, San Jose, CA: 1996.
- [3] M. Abramovici, et. al. *Digital Systems Testing and Testable Designs*, New York, Computer Science Press, 1990.
- [4] H. Michinishi, et. al. "A Test Methodology for Configurable Logic Blocks of a Look-up Table Based FPGA," *Transactions of the Institute of Electronics, Information and Communication Engineers*, vol. J79D-I, pp. 1141-1150, 1996.
- [5] C. Stroud, et. al. "Built-In Self-Test of Logic Blocks in FPGAs (Finally, a Free Lunch: BIST Without Overhead!)," *IEEE VLSI Test Symposium*, 1996.
- [6] W. K. Huang and F. Lombardi, "An Approach for Testing Programmable/Configurable Field Programmable Gate Arrays," *IEEE VLSI Test Symposium*, 1996.
- [7] X. T. Chen, et. al. "A Row-Based FPGA for Single and Multiple Stuck-At Fault Detection," *IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, 1995.
- [8] N. Shnidman, W. H. Mangione-Smith, and M. Potkonjak, "Fault Scanner for Reconfigurable Logic," *Advanced Research in VLSI*, Ann Arbor, MI, 1997.
- [9] J. Rose, et. al. "Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency," *IEEE Journal of Solid State Circuits*, vol. 25, pp. 1217-1225, 1990.
- [10] W. S. Carter, et. al. "A User Programmable Reconfigurable Logic Array", *Proceedings of the Custom Integrated Circuits Conference*, pp. 233-235, 1986.
- [11] W. R. Moore, "A Review of Fault-Tolerant Techniques for the Enhancement of Integrated Circuit Yield", I, pp. 684-698, 1986.
- [12] D. B. Sarrazin and M. Malek, "Fault-Tolerant Semiconductor Memories", *IEEE Computer*, vol. 17, no. 8, pp. 49-56, 1984.
- [13] S. Kikuda, "Optimized Redundancy Selection Based on Failure-Related Yield Model for 64-Mb DRAM and Beyond", *IEEE Journal of Solid State Circuits*, vol. 26, no. 11, pp. 1550-1555, 1991.
- [14] A. Tanabe, et. al. "A 30-ns 64-Mb DRAM with Built-in-Self-Test and Self-Repair Functions", *IEEE Journal of Solid State Circuits*, vol. 27, no. 11, pp. 1525-1533, 1992.
- [15] J. W. Greene and A. E. Gamal, "Configuration of VLSI Arrays in the Presence of Defects", *Journal of the ACM*, vol. 31, no. 4, pp. 694-717, 1984.
- [16] I. Koren and D. K. Pradhan, "Introducing Redundancy into VLSI Designs for Yield and Performance Enhancement", *International Conference on Fault-Tolerant Computing*, pp. 330-335, 1985.
- [17] C. L. Wey, et. al. "On the Design of a Redundant Programmable Logic Array (RPLA)", *IEEE Journal of Solid-State Circuits*, vol. 22, no. 1, pp. 114-117, 1987.
- [18] N. Hassan and C. L. Liu, "Fault Covers in Reconfigurable PLA's", *Proceedings of the International Conference on Fault-Tolerant Computing*, pp. 166-173, 1990.
- [19] K. N. Levitt, et. al. "A Study of the Data Communication Problems in Self-Repairable Multiprocessors", *Conference Proceedings of AFIPS*, Washington, D. C., Thompson Book, pp. 515-527, 1968.
- [20] N. J. Howard, et. al. "The Yield Enhancement of Field-Programmable Gate Arrays", *IEEE Transactions on VLSI Systems*, vol. 2, pp. 115-123, 1994.
- [21] F. Hanchek and S. Dutt, "Node-Covering Based Defect and Fault-Tolerance methods for Increased Yield in FPGAs", *Proceedings of the Ninth International Conference on VLSI Design*, pp. 225-229, 1995.
- [22] A. Mathur and C. L. Liu, "Timing Driven Placement Reconfiguration for Fault-Tolerance and Yield Enhancement in FPGAs", *Proceedings of the ED&TC 96*, pp. 165-169, 1996.
- [23] J.F. Ziegler, et. al. "IBM Experiments in Soft Fails in Computer Electronics (1978-1994)", *IBM Journal of Research and Development*, vol. 40, no.1, pp. 3-18, 1996.
- [24] T.J. O'Gorman, et. al. "Field Testing for Cosmic Soft-Error Rate", *IBM Journal of Research and Development*, vol. 40, no. 1, pp. 51-72, 1996.
- [25] C. H. Stapper, "A New Statistical Approach For Fault-Tolerant VLSI Systems", *The Twenty-Second International Symposium on Fault-Tolerant Computing*, pp. 356-365, 1992.



Evaluating the Benefits of Hardware Context Switching for Automatic Target Recognition

Jason Leonard and Azra Rashid
Electrical Engineering Department
University of California, Los Angeles
Los Angeles, CA 90095-1594



Evaluating the Benefits of Hardware Context Switching for Automatic Target Recognition

*Jason Leonard and Azra Rashid
Electrical Engineering Department
University of California, Los Angeles
Los Angeles, CA 90095-1594*

1 Introduction

This document reports on an evaluation study conducted at UCLA on the applicability of automatic target recognition (ATR) for context switching reconfigurable computing (CSRC) technology. The evaluation was based on the ATR system developed under the Mojave configurable computing project at UCLA [1]. The Mojave system applies field programmable gate array (FPGA) technology, instead of application-specific integrated circuit (ASIC) technology, in order to achieve higher performance at reduced hardware cost and power consumption.

The remainder of this report is organized as follows. The next two sections provide a brief overview of the UCLA Mojave ATR system. Section 2 introduces the CSRC technology being developed by Sanders, as described in the paper by Scalera and Vazquez [2]. Section 3 discusses the approach and findings of evaluating context switching for ATR. Finally, section 4 presents conclusions.

1.1 UCLA Mojave Automatic Target Recognition System Overview

The objective of automatic target recognition (ATR) is to analyze a digital image or video sequence in order to automatically identify specific types of objects within a scene of interest. Figure 1 presents a high level view of ATR processing. The input to the

system is real-time synthetic aperture radar (SAR) images containing several million pixels. The SAR images pass through a focus-of-attention stage, which identifies regions of interest called *chips*. Chips contain potential targets and therefore must be correlated with a large set of target template pairs, *bright template* the identify pixels of strong expected radar return and *surround template* that identify pixels of strong radar absorption. The target templates are much smaller in image size than chips, and have a binary representation. The correlation results are delivered to a peak detector, which identifies the template and the relative offset at which maximum correlation occurs. Templates with the highest correlation and which exceed a threshold limit, are selected by the peak detection step.

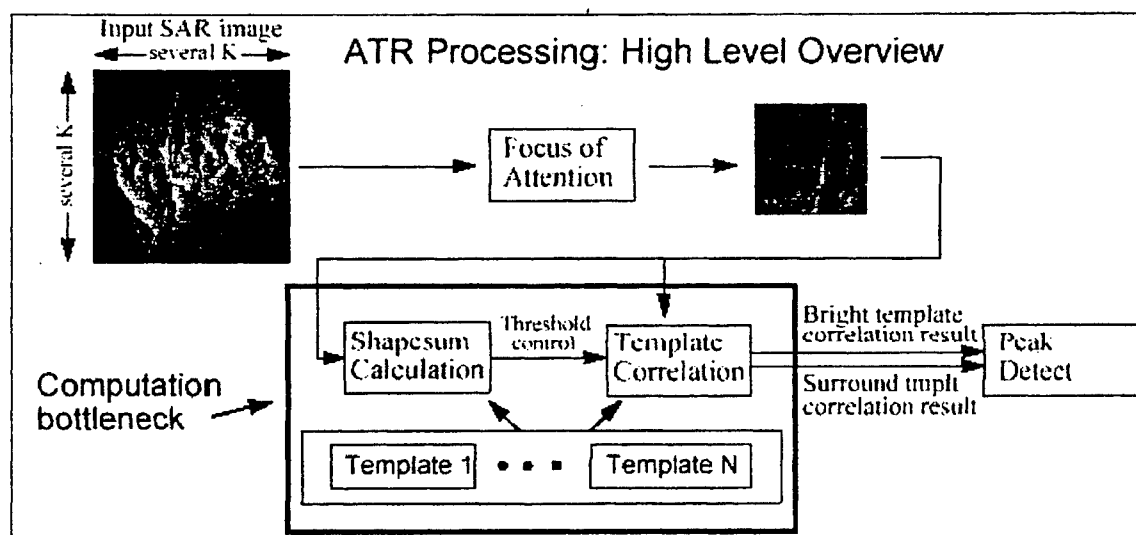


Figure 1. High level overview of ATR processing

Figure 2 shows the template-matching stage in more detail. As mentioned previously, templates occur in pairs: bright and surround. Each template is 8x8 pixels (one bit per pixel). Note that the system studied here uses 32x32 template sizes.

The first step of template matching involves calculating a *shapsum* value. This task is accomplished by adding up the correlation of the bright template against each of the 8-bit pixel planes of the 128x128 chip. Thus, every pixel in the chip maintains a score. The objective here is to account for scene contrast variance (i.e., targets against either bright or dark backgrounds) and thus provide a form of gain control.

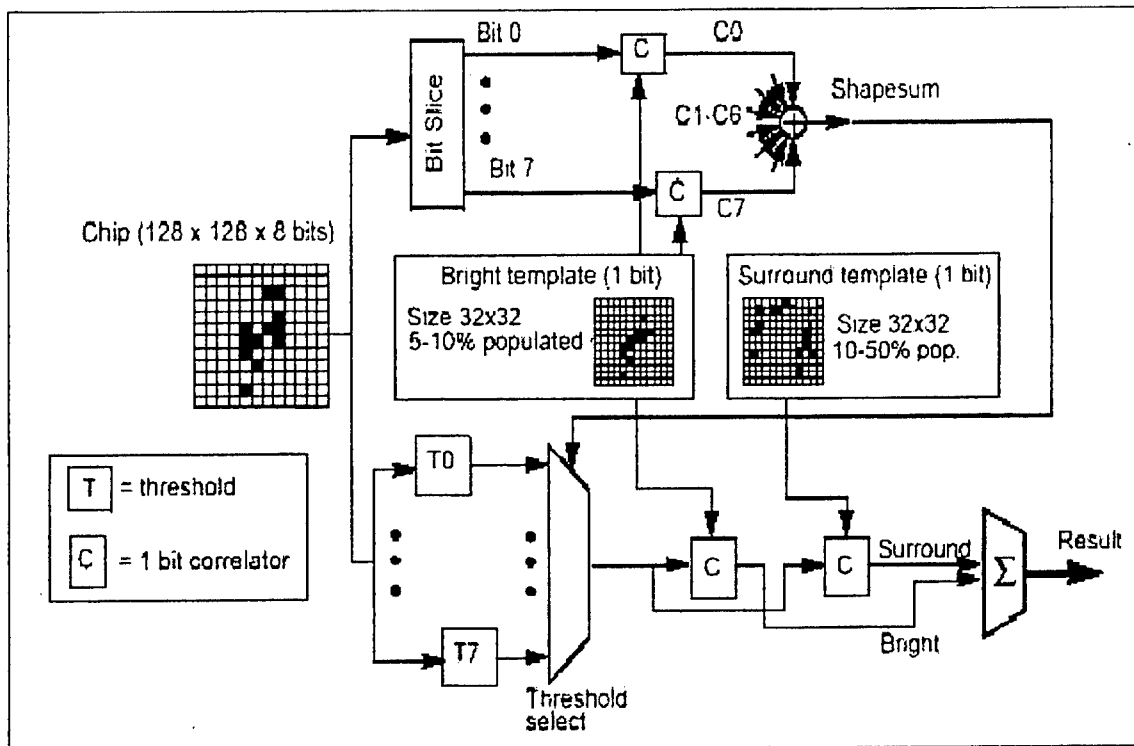


Figure 2. Template-matching stage

The second step involves correlating the actual target templates with the chip. In the original reference design, this function is performed in parallel with step 1. By applying eight threshold values to the chip, eight different binary images are formed. The binary images are correlated with both the bright template and the surround template,

producing eight pairs of correlation results. Finally, the shapsum value is used to select one of eight threshold pairs that will be forwarded to the peak detector.

1.2 ATR Core Design Overview

The ATR core refers to the template-matching logic (Figure 3), which is implemented on a reconfigurable FPGA. It consists of three main modules: main control, correlation logic and control, and FIFO control. The main control module maintains a finite state machine, which provides the timing for all core operations. Each state machine cycle consists of 13 system clock cycles. The correlation logic and control module contains the template matching circuits (RAM-based shift registers and adders), as well as the control circuit that manages the shift operations. The FIFO control provides the template management timing.

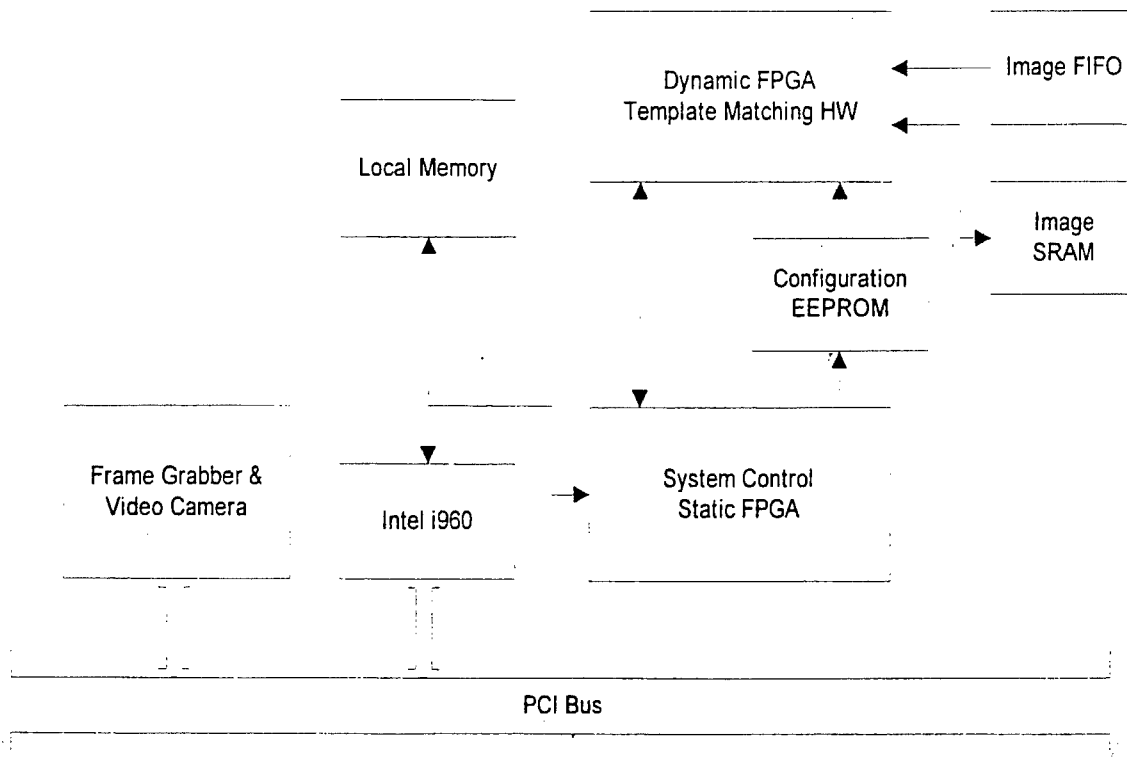


Figure 3. System-level view of the Mojave ATR System

2 Context Switching

The context switching reconfigurable computing (CSRC) FPGA being developed by Sanders, A Lockheed Martin Company, extends commercially available FPGAs to include rapid configuration (context) changes between a number of programmed functions, as well as the capability of data sharing among different configurations [2]. A *context* refers to a specific circuit instance that performs one of possibly many functions or tasks in an application. In a commercial FPGA, reprogramming destroys any resident data, whereas the CSRC FPGA can make the resident data available as input to a subsequent context.

The CSRC device is arranged into 16-bit wide data pipes. Each pipe consists of context switching logic arrays (CSLAs). A single CSLA is capable of processing two 16-

bit words and producing a 16-bit result. The result of one CSLA is available as input to two adjacent CSLAs in the pipe. Thus, a pipe can be used as a data path, where data can flow in both directions. For example, one context could process data from left to right and store its final result in the right-most registers in a pipeline, which can represent an intermediate result of an entire algorithm. The next context picks up where its predecessor context left off by acquiring the intermediate data from the right-most registers and processes it from right to left. The advantage of having a bi-directional data flow mechanism is that it alleviates the need to reroute data from its physical origin in one context to its physical input in the subsequent context.

The CSRC device also supports multi-level routing. Each routing resource consists of 16-bit busses. Level 1 routing consists of three 16-bit busses and interconnects context switching logic cells (CSLCs) within a CSLA. There are 16 CSLCs in a single CSLA. The CSLC is 1-bit computational unit and is composed of carry logic, a 4-input lookup table (CSLUT) containing 16 context switching configuration bits that are multiplexed together, a context switching RAM (CSRam), a context switching flip-flop, and a tri-state buffer. Details of the CSLC structure can be found in [2]. Level 2 routing runs parallel to a pipe. A signal driven onto level 2 routing is available to any CSLA in the pipe via level 3 routing. Level 3 routing runs perpendicular to Level 2 routing. Both level 2 and level 3 routing run the entire width of the CSRC device. Hence, I/O pins are available on these levels.

The CSRC device is formed by stacking data pipes one on top of the other. Corresponding CSLAs on adjacent pipes have dedicated wiring that allows them to propagate their carry bit. This carry chaining allows two adjacent pipes to be bundled

together and be used as a single 32-bit wide data path. Additionally, 16-bit pipes can be broken down into smaller logical pipes. Note that although the CSRC hardware is optimized to break pipes into 4-bit wide, it is capable of supporting n-bits wide pipes.

The key benefits of context switching is provided by sharing data between contexts and being able to switch contexts at fairly high speeds — i.e., significantly faster than the rate at which today's FPGA technology can reconfigure. The CSRC FPGA meets these needs by providing two sharing schemes: Global Sharing and Private/Public Addressable Sharing. Global Sharing is implemented in the I/O cells and CSRam, which will be available in the final CSRC device. Any data written to a CSRam memory is available to all the CSRam elements that are physically collocated among different contexts. Data last written to the active CSRam before deactivation of the current context will be seen by all other collocated CSRams upon the activation of their respective contexts. Thus Global Sharing is simply a common memory element between contexts.

The Private/Public Addressable Sharing Scheme (P/PASS) is used in the CSLCs by the context switching flip-flops (CSFFs). In this type of sharing, each CSFF within each context can be thought of as having a *private* register (Figure 4). The private register belongs to a particular context and therefore can only be accessed by a specific flip-flop within the context. Note that, during normal operation within a single context, the CSFF behaves as a D-flip-flop (DFF). A context can also choose to write its values to a *public* register. Public registers are accessible to all contexts. This allows sharing of data between DFFs.

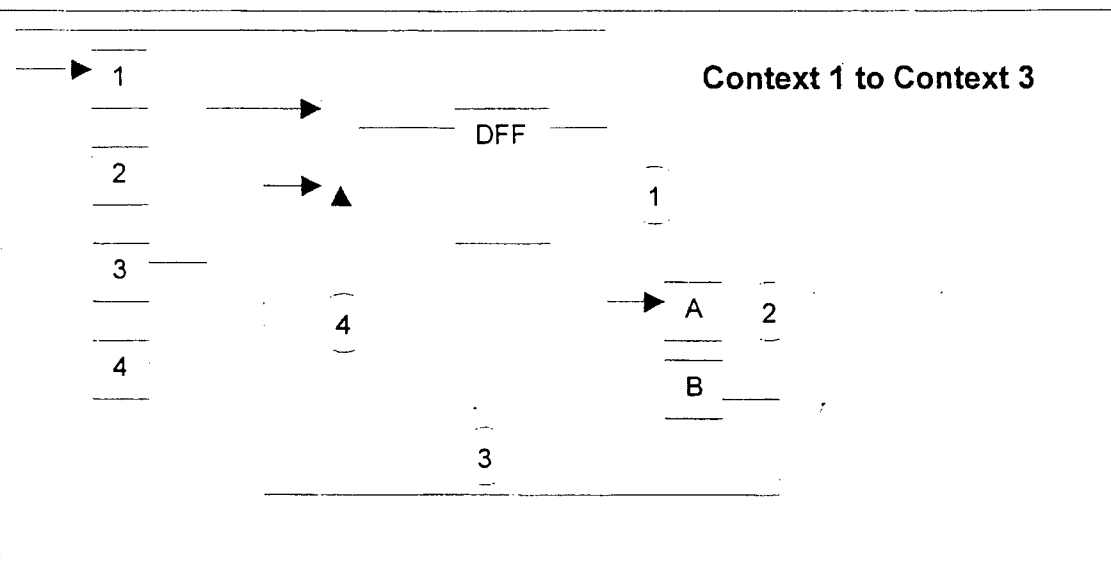


Figure 4. Illustrating Private/Public Addressable Sharing Scheme as context switches between 1 and 3. Dedicated context registers (1-4) get updated every time a context switch occurs. Public sharing registers can be addressed A or B. Upon becoming active, a context can either restore its previous data or load public register. (1) Context 1 saves data to a public register A and its own context register. (2) Determines address of public register the data is stored to. (3) Context 3 loads public address. B determines which public register data can be read from. (4) Determines whether data is restored from a public or private register.

3 Evaluating Context Switching for ATR

3.1 Approach

The first step in evaluating context switching was to partition the existing ATR core design into three circuits or contexts (see Appendix A for VHDL of the partitioned design). The first context contains the logic necessary for shifting the incoming raw image and producing the correlation values at the previous position (i.e., column-to-column mapping). The second context contains the shapsum calculation circuits. Finally, the third context contains the threshold selection and bright- and surround-result summation. Table 1 shows the number of clock cycles per stage.

Table 1. Clock cycle breakdown for ATR contexts

Context	Clock Cycles
1 Shift incoming image and correlation	3
2 Shapsum calculation	6
3 Threshold selection and bright- and surround-result summation	4

The next step was to synthesize the logic for all three contexts, as well as the original full design for comparison. The target FPGA was chosen to be the Xilinx XL4062-3 device. Note that it was necessary to add a small amount of glue logic for the intermediate values in order to disable specific synthesis optimizations. Also, placement constraints were added for the intermediate result logic of contexts 1 and 3. Context 2, because it had the most number of CLBs, but least number of intermediate bits (see Table 2), was likely to have the worst performance and was therefore routed without any placement constraints. The resulting placement mapping provided a “guide” for the other contexts, by constraining the location of shared data.

3.2 Results

Table 2 illustrates the hardware and performance results from the Synopsys Design Compiler logic synthesis tool based on the Xilinx XL4062-3 part. Note that the placement of context designs was constrained to save the intermediate bits in consistent locations for all contexts. The largest intermediate result that needed to be saved was 450 bits, occurring between the shapsum calculation and threshold/summation contexts.

Table 2. Hardware and performance measures of ATR contexts

Context	CLBs	Intermediate Bits Saved	Maximum Clock Frequency
1	44	118	17 MHz
2	1762	24	7.4 MHz
3	816	450	4.5 MHz
FULL	2304	N/A	3.2 MHz

The results suggest that it may not be beneficial to keep a separate context 1. A better approach would have been to include it with context 3, thereby dividing the processing into two steps (6 and 7 clock cycle steps, respectively). If the 7.4 MHz clock speed of context 2 can be achieved by the combined contexts of partitions 1 and 3, then greater than a factor of two speed improvement would be achieved over the full design (3.2 MHz) for a 4062 part (assuming a fast context switch). Additionally, the reduction in CLB count would make managing of larger templates much easier. Another approach would have been to pipeline context 3 in an effort to achieve a factor-of-two clock speed improvement.

It is also evident from the results that a more efficient placement of context 3 is required. As previously mentioned, the method used here was to use the 24-bit placement mapping result of context 2 as a "guide" for the Xilinx XACT M1 tools. This was clearly insufficient, despite the fact that context 3 had only half the CLB count as context 2. However, at the time, the Xilinx M1 tools were not suitable for this type of application (i.e., high number of constrained bits without any repeated structure to the logic).

Data-Specific Number Factoring on Context Switching Configurable Computers

Hea Joung Kim

kimmer@icsl.ucla.edu

Electrical Engineering Department
University of California, Los Angeles
Los Angeles, CA 90095-1594

Introduction

As part of the Context Switching Configurable Computing project, we have been investigating the opportunity apply configurable computing technology to the problem of number factoring. We will begin by briefly reviewing why number factoring is an important problem, as well as the fundamental characteristics of the dominant factoring algorithms.

Encryption is an integral part of modern secure communications systems. The RSA (Rivest, Shamir, Adleman) public-key encryption algorithm, which is currently the most secure scheme in wide use, relies upon the fact that factoring large numbers is extremely difficult^[1]. Since the RSA encryption algorithm was introduced, there have been many attempts at cracking the keys for the RSA encryption algorithm. In order to break the RSA algorithm, large numbers have to be factored. Number factoring is a difficult problem that has been studied for thousands of years, and there have been a number of algorithms for factoring large numbers with actual implementations in C programs. With the knowledge of the C program implementations, a proposal is described to implement a factoring algorithm in hardware. The following sections briefly describe the RSA encryption algorithm. A description for factoring large numbers follows. Lastly, a hardware implementation is detailed along with the performance analysis.

The RSA encryption algorithm requires two large prime numbers p and q . For maximum security, p and q are chosen to be of equal length. After the appropriate p and q are chosen, they are verified to be primes with the primality test. With p and q , the number $N=(p-1)*(q-1)$ is generated. Using the numbers p and q , the encryption key (e) is chosen randomly such that the encryption key (e) and the number N are relatively prime. Relatively prime means that two numbers share no factors in common other than 1. Relative prime can be described by the equation $(e=1 \bmod N)$. The relative prime requirement guarantees that ' e ' is prime as long as N is not prime. Using these relations and numbers, another key can be generate using the equation $d=1/e \bmod N$. The number N is shared information while e and d form the public and private keys . Using the private and public keys, a message can be encrypted and decrypted in the following manner:

1. Take a message M and divide it into blocks smaller than N .
2. The encrypted message will be $C=M^e \bmod N$ (where e is the public key).
3. After transmission, the decryption equation is $M=C^d \bmod N$ (d is the private key).

Since the publication of RSA for encryption, many mathematicians have worked to expedite the process of factoring very large number (i.e. RSA-129 is a 129 digit decimal number composed using a 64 digit and 65 digit prime numbers). With a fast factoring algorithm, the number N can be factored to give p and q ; thus, the encryption keys can be found to decipher any encrypted message. In order to find the two prime numbers, the equation $(x^2 = y^2 \text{ modulo RSA129})$ has to be satisfied by finding the appropriate x and y . If x and y are found, there is a good chance that the greatest common divisor (gcd) of $x+y$ and RSA129 is a prime number (not equal to 1 or RSA129). Thus, a prime factor of RSA129 is found and the other prime number is easily found. There are quite a few number factoring algorithms. The fastest known algorithm for large numbers is the number field sieve [2]. The quadratic sieve is the next best known algorithm [3]. Both algorithms use a sieving method to find $(a = b^2 \text{ modulo RSA129})$ where 'a' is the quadratic residue if an integer 'b' can be found. The 'a' is usually made up of many small primes plus 0,1, or 2 other primes less than 2^{32} (i.e. $a = p_1^{n_1} * p_2^{n_2} \dots * p_i^{n_i}$), and there are $(\text{RSA129})/2$ number of 'b's'. In the following sections, a hardware implementation study of the sieving algorithm is presented.

The sieving process traverses an entire array of zeroed locations using prime numbers for access strides and adds the logarithm of the primes to the value on the array location. The logarithm of a prime can be used because 'a' is composed of multiple primes raised to some number. After the sieving is done, the array is scanned for numbers that exceed a threshold, which is chosen high enough to produce an acceptable false alarm rate and yet low enough to guarantee no false negative results. Usually, a high number in the array means that the 'a' is highly composed of many prime numbers, i.e. is a highly composite number. These numbers are of interest and are checked to see if they are indeed composed of many primes. After many repetitions, the 'a's' with even powers of the prime combinations along with the 'b's' are used to satisfy the $x^2 = y^2 \text{ modulo RSA129}$ equation. Even powers of prime numbers (a's) make the x^2 and the 'b's' are already a square. These results could bring about a non-trivial prime number for $\text{gcd}(x+y, \text{RSA129})$.

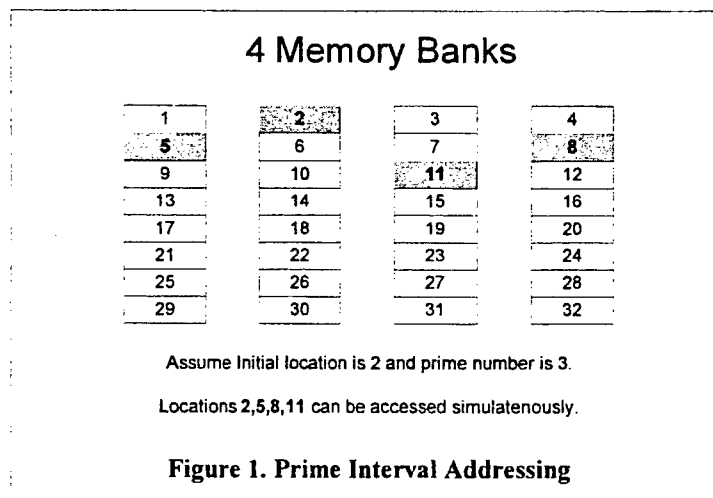
The multiple polynomial quadratic sieve (MPQS) [4] has been realized using a C program [5], which we have used as the basis for our analysis. After profiling several executions, it became apparent that the time spent in sieving and thresholding the array is approximately 80% of the CPU time.

Table 1. Percentage of CPU time used on the Sievers of MPQS.c

Function	Percentage of CPU time used
Big_prime_siever	29.9
Low_prime_siever	24.6
Med_prime_siever	12.7
First_ge (find locations that exceed threshold)	6.1

In the MPQS.c program, the sieving operations are divided into three sievers (low for primes $\leq \frac{1}{2}$ of sieve array, medium siever for $\frac{1}{2}$ of sieve array \leq prime $<$ sieve array, and big for prime \geq sieve array) depending on the size of the prime numbers. The sievers are iterative loops that add the logarithms of the set of prime numbers to the sieve array locations. The roots of the quadratic residue equation initially determine the locations of the sieve array. The logarithms of primes are integers and are defined by 8-bit numbers, thus making the addition cheap and efficient in hardware. The read-add-store loop is not limited by the addition of the logarithm of the prime but by the address generation for read and write and bandwidth to the sieve array.

A closer look at the low prime siever shows that the memory locations that the sievers hit are multiples of prime numbers. As a consequence, a special memory-addressing scheme can be used to address sieve array. The prime multiple addresses allow for parallel addressing with unlimited scalability,



as memory bank conflicts can be trivially avoided using standard powers-of-two memory design. The limitation comes from the input/output pins of a custom or programmable chip. These types of optimizations allow for even further speed up of the sieving algorithm. The FIRST_GE function can also be sped up with the parallel addressing of the memory location with relative ease.

The memory interleaving scheme presents itself well for the implementation of the sievers in hardware. Furthermore, the application will not be limited by the chip's critical path but rather the memory access time. An FPGA may be sufficient to implement the sievers instead of using an ASIC.

After considering each of these factors, we have concluded that there is definitely a performance opportunity present in implementing the sievers in hardware. With the speed up of the sievers, it is easy to realize a performance increase of 4 for the MPQS.c program. These advantages are the motivations for implementing the number factoring sievers in hardware.

FPGA Implementation

The functions within the MPQS.c program that could speed up the entire factoring task are the sieving processes. The sievers are divided into three types depending on the size of the prime number relative to the sieve array. If the prime number is less than half of the sieve array, the low prime sieve is executed. Otherwise, if it is greater than half of the sieve array and less than the sieve array, the medium sieve is executed; else, the big prime sieve is executed. Table 1 shows the percentage of CPU time used by each of the sievers.

The key advantage of using a FPGA for the sieving process is due to the addressing scheme shown in Figure 1. The fact that the addresses are incremented by intervals of prime numbers makes it possible for the FPGA to process 4 sieving operations in parallel. Furthermore, if there are more IO (input/output) pins available on an FPGA more parallel processing could be done. This is simply due to the prime stride addressing.

The other interesting fact about the sieving process is the fact that an ASIC will not be beneficial at faster clock speeds due to the memory access time limitation. With the next generation FPGA technology, it will be possible to clock many designs at a 125MHz or above, and this certainly will be true of a simple design such as the sieve we envision. At this frequency, the main bottleneck is the memory access time. As shown below on the operation analysis, the main operations of the sievers constitute a read from memory, an 8-bit addition, and a write back to memory. The critical path on chip will be determined by the address generation time, which is done simply by adding a prime number to the current address. The addresses are at most 22 bits for a 4Mbyte SRAM.

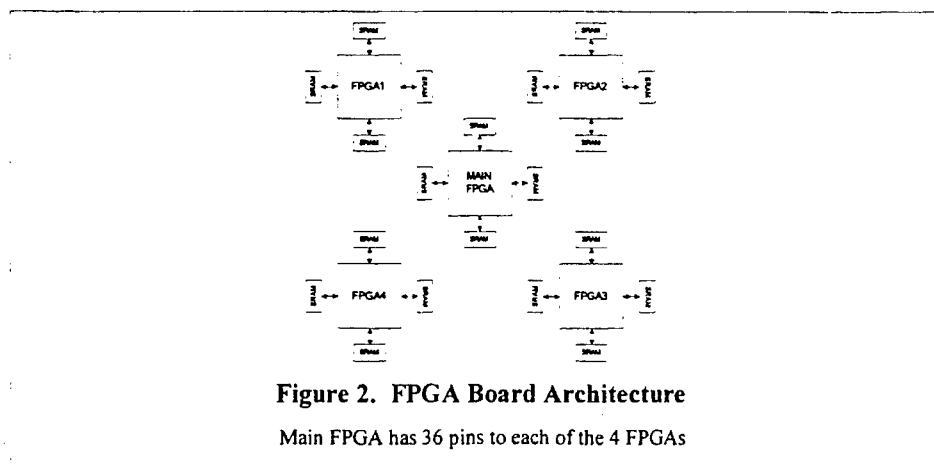


Figure 2. FPGA Board Architecture

Main FPGA has 36 pins to each of the 4 FPGAs

Figure 2 shows the 5 FPGAs that can be used for sieving. The layout of the FPGAs matches the current design built by UCLA as part of the Mojave Configurable Computing Project. The initial analysis will be performed on a single FPGA (the Main FPGA). Each FPGA has 4 SRAMs accessible with independent addresses. In the following sections, the performance of MPQS.c is compared to that of the FPGA using some estimated numbers. The exact numbers will be presented after the entire system is built and tested.

Pseudo code for low_prime_siever

```
sieve_rsl = sieve;           // rsl = sieve array size (i.e. 3,000,000)
                             // sieve is first location address of sieve array
for (i = ibeg - iend; i <= 0; i++) {
    root2= rootsi->r1 + rootsi->r2;
    p = *pi++;                // increments prime
    sievel = rootsi->r1;       // offset address
    do {
        sievel[root2] += loc_logp;
        sieve_rsl[root2] += loc_logp;
        root2 += p;
    } while (root2 < rsl);
}
```

Low_siever with the following parameters:

Root sieve length (rsl) =3,000,000
 Begin count (ibeg) =16
 End count (iend) =19
 Prime =101
 LogPrime (logp) =4

Outer loop (iend-ibeg) => 3 loops

 Inner Loop (rsl/prime) = 3,000,000/101 ~ 30,000

 instructions per Inner loop (address 2 locations, add logp, write back, and inc addr)

FPGA Implementation analysis,

3 (for loop) x 3000000/101 (do while loop) = 90,000 sieve operations

FPGA can perform 4 sieve operations (4 addresses per clock) in 3 clock cycles.

At 33MHz, (90,000 * 4/3 * 30ns) ~ 3.6ms.

Low siever running on SPARC20	120ms
Low siever in an FPGA (@33MHz)	3.6ms

There is a factor of 40 speed up that can be achieved by using an FPGA running at 33MHz. At 125MHz, the sieve operation shown above can be performed in 0.96ms, which is a speedup of 125 times over the SPARC.

Number Factoring on a Context Switch FPGA

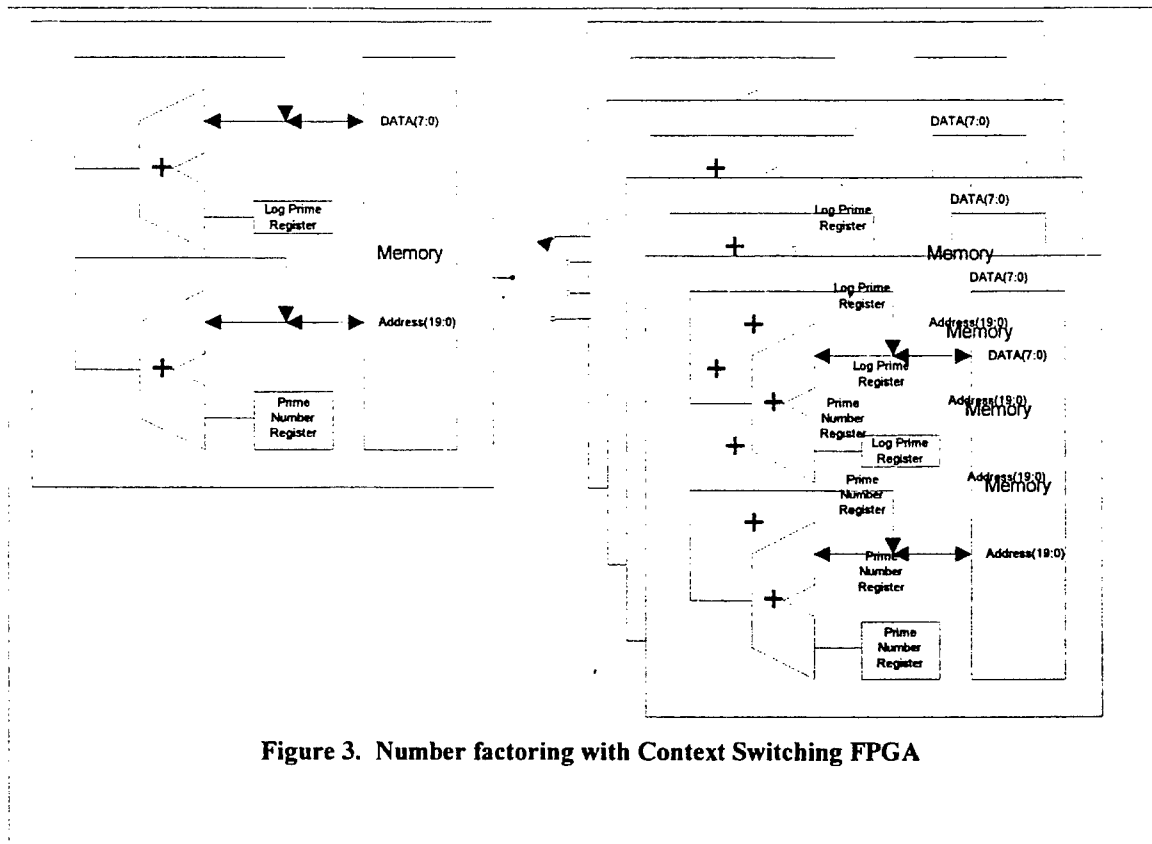


Figure 3. Number factoring with Context Switching FPGA

With a context switching FPGA, the fact the multiple iterations of adds for the log prime to the sieve array and address generation adds using the prime can be exploited. The context switch FPGAs can be reprogrammed for each new prime and log of the prime to generate a specific circuit with optimized adders for this task. With the data specific adders, the additions can be performed at the speed of a look up table. The performance gain using a context switch part will be significantly faster.

Texas Instruments TMS320C6201 DSP Implementation

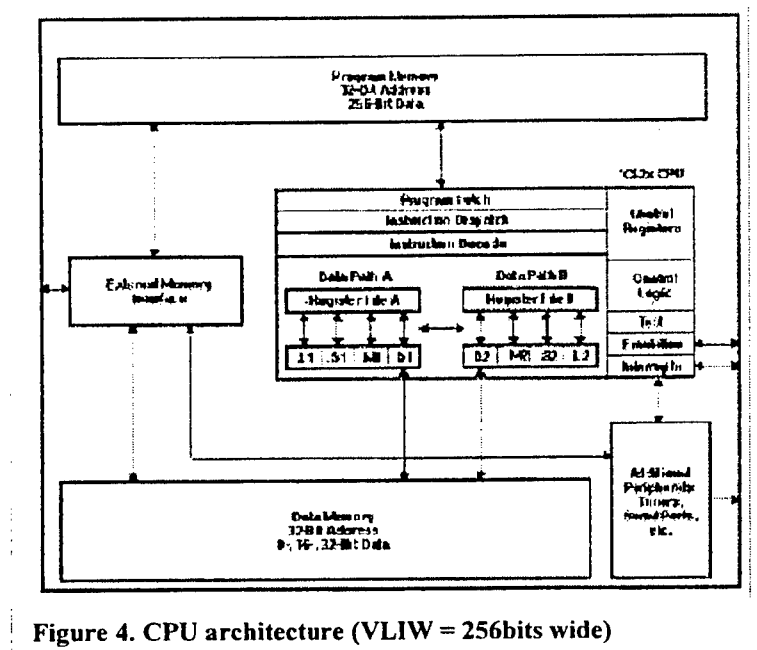


Figure 4. CPU architecture (VLIW = 256bits wide)

Another approach to implementing the sievers is to use a VLIW (very-long instruction word) architecture. An interesting CPU for the sievers is the TMX320C6201 by Texas Instruments. The core of the CPU has 8 execution units as shown in Figure 3. The reason that this CPU is selected for this study is due to its two data addressing units (.D1 and .D2) because the sievers performance scales linearly with the number of data locations that it can address concurrently. The key data that would be of interest is how many locations the CPU will be able to bring into the DATA memory using the external memory interface for concurrent execution.

Using the TMX320C6201, it is quite clear that the address generation and the addition of the logarithm of primes will not be an issue. Four of the other 6 units can be used to perform the 2 logarithm of prime additions and 2 address generation additions. Given that the processor operates at 200MHz, the additions will not be the bottleneck. The memory accesses will be the most important system factor.

In a TMS320C6201 (vector processor with 8 operation units running at 200MHz), 8 instructions that can be executed are 4 arithmetic operations, 2 multiplies, and 2 data addressing operations.

Assume a cache hit (data available in the core DATA memory shown in Figure 3) and ignoring the latency associated with pipelining.

Given that every sieve operations needs:

1. prime number interval addressing unit
2. simple add
3. increment root by prime
4. write back.

With 2 addressing units

1. bring in 2 sieve locations, do 2 adds, increment 2 addresses by prime (using the 4 arithmetic units)
2. write back 2 sieve locations

Two clock cycles are needed to do one sieve operation. Thus, 100 million sieve operations per second can be performed. The TMS320C6201 will perform the same above operation in $(5\text{ns} * 90,000) = .45\text{ ms}$

The TMS320C6201 with some poor assumptions show that it could perform very well. However, if each data access requires an operation from the external memory functional unit, the memory access time will dictate the overall performance of the system. Thus, instead of a sieve operation being performed in 5ns, it may be 100ns (two accesses to DRAM). An access time of 100ns translates to 9ms for the sieve operation described above. The generation of TMX320C6201 code and testing will show the actual performance achievable.

Development (timeline)

Hardware Development/Simulations/Test/Verification Schedule

Q3'98	Q4'98	Q1'99	Q2'99	Q3'99
Demo Sievers on Mojave Board <ul style="list-style-type: none"> • VHDL Hardware • Simulation • Integration into PCB 	Demo on small version of CSRC first Board <ul style="list-style-type: none"> • Context switch base adders 	TI TMX320C6X Evaluation <ul style="list-style-type: none"> • Simulation results on actual tests to see memory performance 	Code optimization for transfer to Complete CSRC board	Demo on Complete CSRC System <ul style="list-style-type: none"> • Context switch base on log prime and prime intervals of address

¹ L. Rivest, A. Shamir, L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM.*, v. 21, 1978, pp. 120-126.

² A. K. Lenstra, H. W. Lenstra, Jr., M.S. Manasse, J.M. Pollard, *The number field sieve*, pp. 11-42; extended abstract: Proc. 22nd Annual ACM symp. On Theory of Computing, Baltimore, May 14-16, 1990, 564-572.

³ J.L. Gerver, "Factoring large numbers with a quadratic sieve," *Math comp.*, 41 (1983), 287-294.

⁴ R.D. Silverman, "The multiple polynomial quadratic sieve," *Math. Comp.*, v. 48, 1987, pp. 329-339.

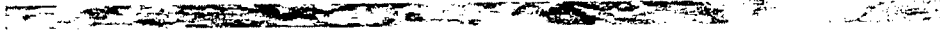
⁵ A. K. Lenstra and M.S. Manasse, *Advances in Cryptology, EUROCRYPT '89*, Lecture Notes in Computer Science, vol 434, Springer-Verlag, 1990, pp. 355-371.



Suitability of Bin Packing for Context-Switching Configurable Computing Technology

Itai M. Pines
itai@icsl.ucla.edu

Electrical Engineering Department
University of California, Los Angeles
Los Angeles, CA 90095-1594



Technical Paper #98-3

Suitability of Bin Packing for Context-Switching Configurable Computing Technology

Itai M. Pines
The Department of Electrical Engineering
The University of California
Los Angeles, CA 90095-1594
itai@icsl.ucla.edu

1. Introduction

Configurable computing systems [1] can exceed the performance of both software and custom hardware provided they exploit opportunities to customize the programmable hardware to a specific problem instance rather than the general problem [2]. The key to exceeding the performance of an ASIC is to use a circuit that is specialized to both the application and a particular problem instance, i.e. data set. We refer to such a device as a problem-specific integrated circuit, or a PSIC. PSICs are a developing technique for solving certain computationally challenging problems. By creating problem-specific hardware it may be possible to significantly reduce the number of operations required to find a solution. Specializing hardware to data is one of the unique capabilities of configurable computing systems and has been well investigated [3,4]. Traditional attempts to solve NP-complete problems in hardware are general-purpose in the sense that they do not consider the specific instance of the problem to be solved. While software solutions allow for some specialization, they are constrained by a fixed set of hardware resources. PSICs are a particularly aggressive form of data specialization in that they incorporate the entire data set and produce the solution to a specific instance of a problem. Thus, any specific PSIC is run once, its result noted, and then the circuit is discarded. The problem-specific hardware is implemented on an FPGA so that it can be reconfigured for each new instance of the problem requiring solution. Clearly, such an approach is not economically viable with ASIC technology.

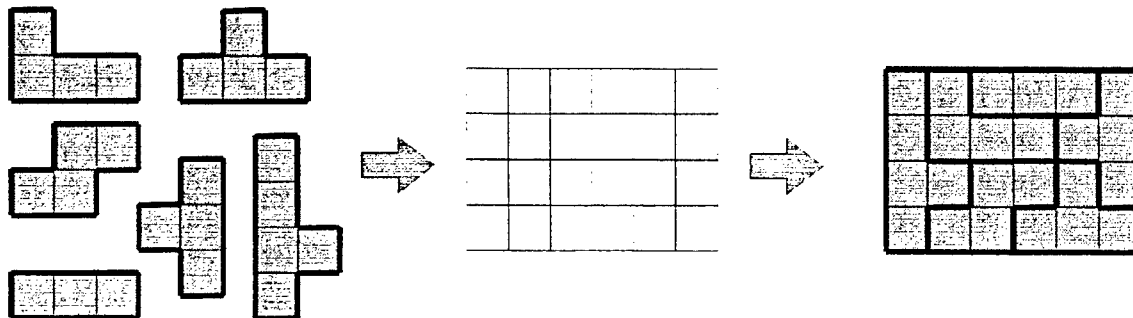


Fig 1 – Illustration of Bin Packing problem.

The problem of Bin Packing (Fig 1) – deciding whether a set of predefined shapes will fit into a two dimensional space without overlapping – represents a typical NP-complete problem and has applications in VLSI layouts [5], image processing [6], and hardware resource allocation [7]. In addition, other NP-complete problems can be transformed into Bin Packing problems and solved. Bin Packing is NP-complete, therefore the only known method guaranteed to find a solution is to search the full set of all possible solutions. Software solutions for Bin Packing exist which attempt to reduce the complexity of the problem and speed up finding a solution [8], however they are still limited by the effectiveness of the general-purpose processors on which they are implemented. We hypothesized that by using configurable hardware, a more efficient method of solving Bin Packing problems might be produced. By building a specialized circuit dedicated to a particular solution space and set of desired objects, a reduction in problem complexity would allow for a more rapid solution. This report summarizes our study methods and conclusions.

2. Hardware Only Implementation

In terms of hardware implementation, there are two basic models available: independent hardware or coprocessor hardware. In the independent hardware model, the PSIC is a standalone circuit requiring only a clock and an interface to read out the solution when it is discovered. The advantage of this approach is that all of the problem complexity is placed into one construct, offering the maximum opportunity to specialize the circuit to the data set. In addition, because there is no need to burden the central processor during run time, it is possible to run several PSICs simultaneously, or decompose a large problem into manageable sub-elements and solve them in parallel. Also, because it has little responsibility for performance, the requirements for processor speed are minimal. The disadvantage of the independent hardware model is that all aspects of the problem must be solved by the PSIC, including functions that may be better implemented by a general-purpose processor or DSP. In the coprocessor hardware model the PSIC is explicitly connected to a central processor and serves to accelerate the solution of a problem by easing the workload of the processor and allowing it to solve the problem in fewer steps or handle larger problems than would otherwise be feasible. This partitioning simplifies the design of the PSIC and allows certain potentially difficult hardware functions, such as arithmetic, to be implemented in software. Unfortunately, this approach lessens the extent to which the PSIC

is specialized to the data set. Either implementation has the potential to improve performance; which implementation strategy is chosen depends on the nature of the problem to be solved.

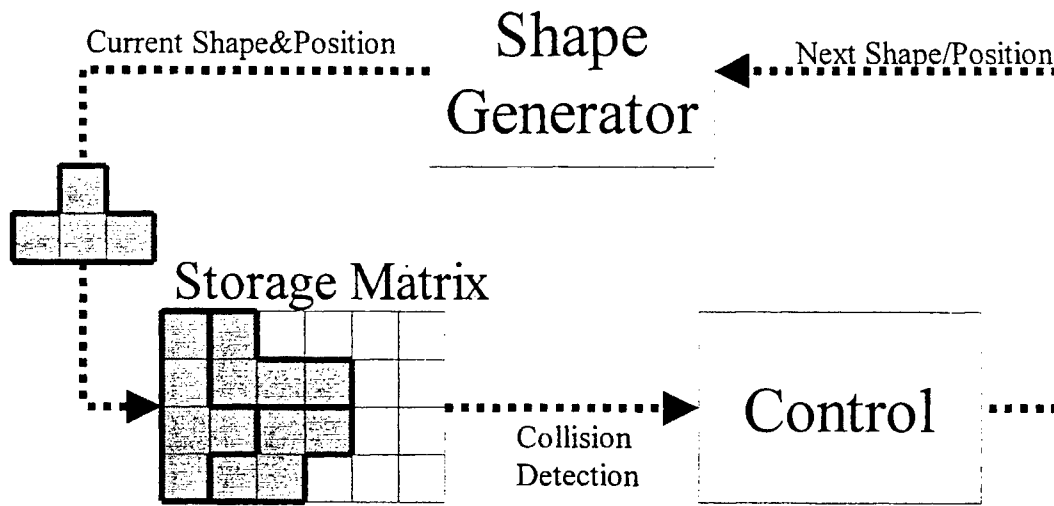


Fig 2 – Independent Bin-Packing Processor Implementation

We first investigated the independent hardware approach, which created a highly specialized autonomous circuit that searches the solution space until a solution is found or proven infeasible (Fig 2). The shape generator is hardwired to produce the necessary shapes in the appropriate positions to the storage matrix. The storage matrix tests the new object and location for conflict and if none exists, adds the new object. Given the explicit knowledge of the shapes to be placed, the collision matrix can be optimized so that it only looks for collisions where they have the potential to occur. This optimization can reduce the amount of routing required. The control block determines the sequence of objects and locations and will notify the other elements in the event that a previous placement proves untenable and the object must be removed. In such a situation, the control block explicitly notifies the collision matrix that the next object produced is to be removed, then on the next cycle returns to the previous object and attempts to find another location for it. This back-tracking is implemented primarily with counters; a counter overflow indicates that an object cannot be placed and the previous object must be removed and placed elsewhere. If the first object cannot itself be placed, the control block halts execution and notifies the user that no solution exists. Alternatively, when the last object in the stack is successfully placed, the control block ceases operation and notifies the user that a solution has been found. The exact solution can then be read from the control counters. The timing control in this circuit is distributed; each element begins to operate as soon as the previous stage completes.

Since there is no need to synchronize the internal workings beyond maintaining data dependencies, the system can run at the maximum speed afforded by the combinational logic. The low-level operations of generating objects, testing placement and maintaining state can be optimized to the particular data set. In addition, a straightforward CAD optimization can group small objects into sensible sub-blocks using a greedy algorithm and reduce the overall number of operations significantly.

This approach has the advantage of hardware efficiency (only the minimal hardware required to solve the problem would be produced), and autonomy, (the process required no oversight by a central processor). In order to utilize the FPGA the problem-specific hardware must undergo place and route before being loaded onto the FPGA. Unfortunately the time required to place and route the hardware for a configurable computing platform can measure in hours and is so large that it rendered any benefit to solution time insignificant. From this result we learned that a viable solution would require designing for automatic place and route, using modules that could be produced in advance and interconnected on the fly. Ideally such a mechanism would involve building a circuit out of predefined basic blocks with implicit routing.

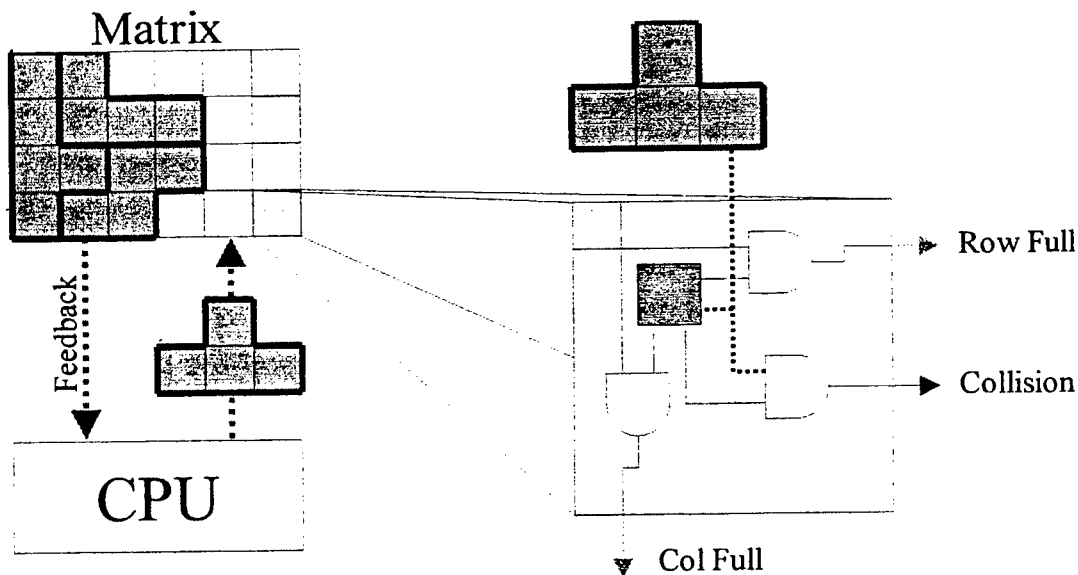


Fig 3 –Bin-Packing Coprocessor Matrix Implementation

3. Coprocessor Implementation

The results of our investigation of a hardware only solution led us to our second approach, involving a matrix of cells which would function as a coprocessing element and speed up solution by providing useful feedback to the central processor (Fig 3). In this model the processor implements all of the functionality of the control and shape generation modules. The matrix implements storage of objects and notifies the processor if an attempted placement will result in a collision. In addition, the matrix produces signals to notify the processor if an entire row or column has been filled. The row and column full signals allow the system to skip over attempts to place objects where they cannot possibly fit, thereby reducing solution time in a manner not available to a more general solution.

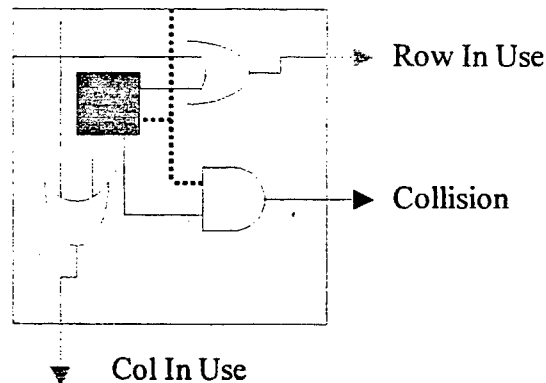


Fig 4 –OR Gate Cell Implementation

Since the matrix could be reconfigured it would be possible to substitute in certain specialized modules to assist in various phases of the solution. One example would be the OR gate cell (fig 4), which would allow for tighter grouping of placed objects by notifying the processor of which rows were in use at any point during the solution. Using these cells in the initial stages of a solution would allow the processor to quickly compare different groupings of several of the objects to see which grouping was most compact and most rectangular. By building objects into subsets with high efficiency the complexity of the problem could be reduced and larger problems could be solved. Since the hardware can be reconfigured at run time it would be possible for the processor to swap in specialized cells to assist in any difficult phase of solving the problem without losing the results already achieved, thus directly leveraging multiple context-switching technology.

The regularity of the circuit made automatic routing a possibility and promised to map well to an FPGA, as well as being inherently scalable. Unfortunately this approach abrogated most of the advantages of a problem-specific solution, the hardware became general-purpose and required explicit oversight by a general-purpose processor. Since a central processor was required we could only fairly compare this solution to a pure software solution. In an equivalent software solution, all of the metrics could be simply implemented with a set of constantly updated registers, which could be located in an additional memory module added in place of our proposed FPGA. It became clear that the PSIC solution offered diminishing benefit over a software solution, which would gain more from extra memory than from our customized hardware.

4. Conclusions

While these results are somewhat discouraging, they provide useful information about what type of problems are more likely to be effectively solved using configurable hardware. The first observation is that the solution must be automated in such a way as to circumvent the need for general-purpose place and route CAD tools. To achieve this goal the solution circuit must be built out of predefined basic blocks with implicit interconnect routing. One excellent example of such a problem is Boolean Satisfiability, currently being investigated by members of our research group[9]. Unlike Bin Packing, Boolean Satisfiability problems can be expressed in a netlist format which can be translated directly into a two dimensional circuit layout built of basic blocks and simple routing. The second problem we encountered was the need for sufficient specialization to justify the cost of using a hardware solution. Here too, the Boolean Satisfiability problem is inherently specialized in the design of the solution circuit as each circuit maps directly to a unique Boolean equation to be solved. In conclusion, while Bin Packing appears to be unsuitable for PSIC implementation, there is compelling evidence that the equally intractable problem of Boolean Satisfiability can be solved effectively using the PSIC methodology.

6. Acknowledgements

This work was supported by the Defense Advanced Research Projects Agency of the United States of America under contract DAB763-95-C-0102 and subcontract QS5200 from Sanders, a Lockheed Martin company, as well as NSF CAREER award #9734132

7. Bibliography

- [1] J. Villasenor, W. H. Mangione-Smith, "Configurable Computing," *Scientific American*, June 1997.
- [2] J. Villasenor, B. Schoner, K.-N. Chia, C. Zapata, H. J. Kim, C. Jones, S. Lansing, and W. Mangione-Smith, "Configurable Computing Solutions for Automatic Target Recognition," *Proc. of the IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, 1996.
- [3] M. J. Wirthlin and B. L. Hutchings, "Improving Functional Density Through Run-Time Constant Propagation," *Proc. of Field Programmable Gate Arrays*, Monterey, CA, 1996.
- [4] J Cong and J. Peck, "On Acceleration of the Check Tautology Logic Synthesis Using an FPGA-based Reconfigurable Coprocessor," *Field-Programmable Custom Machines*, 1997.
- [5] H. Murata, K. Fujiyoshi, S. Nakatake, Y. Kajitani, "Rectangle-Packing Based Module Placement," *International Conference on Computer Aided Design (ICCAD)* 1995
- [6] R. J. Petersen and B. L. Hutchings, "An Assessment of the Suitability of FPGA-Based Systems for use in Digital Signal Processing," in *Field-Programmable Logic and Applications*, W. Luk, Ed. Oxford, England: Springer, 1995, pp 239-302.
- [7] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*: W. H. Freeman and Company, 1979.
- [8] J. Silva and K. Sakallah, "GRASP-A New Search Algorithm for Satisfiability," *IEEE ACM International Conference on CAD-96*, pp 220-227, Nov. 1996.
- [9] A. Rashid, J. Leonard and W. H. Mangione-Smith, "Dynamic Circuit Generation for Solving Specific Problem Instances of Boolean Satisfiability," *Field-Programmable Custom Computing Machines (FCCM)* 1998.

Algorithms for Runtime Tolerance of Interconnect Faults on Diverse FPGA Architectures

John Lach, William H. Mangione-Smith, Miodrag Potkonjak[†]
Electrical Engineering Department & Computer Science Department[†]
University of California, Los Angeles
Los Angeles, CA 90095-1594

Algorithms for Runtime Tolerance of Interconnect Faults on Diverse FPGA Architectures

John Lach⁺, William H. Mangione-Smith⁺, Miodrag Potkonjak⁺⁺

Departments of Electrical Engineering⁺ and Computer Science⁺⁺

University of California, Los Angeles

Abstract - Fault-tolerance is an important system metric for many operating environments, from automotive to space exploration. The conventional technique for improving system reliability is through component replication, which usually comes at significant cost: increased design time, testing, power consumption, volume, and weight. We recently developed an approach for tolerating logic faults that capitalizes on the unique reconfiguration capabilities of FPGAs. The algorithm has been expanded to tolerate interconnect faults and can be implemented on a variety of FPGA architectures. Although some architectural features allow for a more efficient implementation, high levels of fault-tolerance with low timing and resource overhead can be achieved on diverse architectures.

1 General Algorithm

Taking advantage of the flexible and inherently redundant nature of FPGAs, a low overhead fault-tolerance algorithm has been developed capable of tolerating faults at runtime with minimal system downtime and no end user CAD tool requirements. Assuming a detection, localization, and diagnosis¹ of a fault, a configuration of the design can be loaded that does not utilize the faulty resource(s). The alternate configurations have been previously generated by the CAD tools at design-time and are available in memory. The proper configuration is then activated based on the location of the faults. The previously prepared configuration need only be applied to the device, thereby not requiring substantial system downtime or the end user to have access to CAD tools. Therefore, the algorithm, and even the very existence of an FPGA in the system, remains transparent to the user. Previous algorithms achieved such runtime fault-tolerance exclusively for logic faults [1], but the algorithm has been expanded to include interconnect faults and has been applied to a variety of FPGA architectures.

We propose partitioning the physical design into a set of tiles. Each tile is composed of a set of physical resources (i.e. logic blocks and interconnect), an interface specification which denotes the connectivity to neighboring tiles, and a netlist. Logic and local interconnect reliability is achieved by providing multiple configurations of each tile, each of which does not use certain resources within the tile. Furthermore, by using immutable tile interfaces, the effects of swapping a tile configuration do not propagate to other tiles, thereby making each tile independent and reducing the storage overhead². Any paths that cross tile boundaries can be made reliable by reserving other inter-tile interconnect to be used as spares.

This approach has three main benefits compared to redundancy-based fault-tolerance: very low overhead, the option for runtime management, and flexibility. The overhead required to implement this fine-grained approach, which can be measured in both physical resources on the FPGA (logic blocks, I/O blocks, and routing) and circuit performance, is extremely low compared to redundancy. Runtime management can be a very valuable feature of a system, particularly for mission-critical applications. This fault-tolerance approach handles runtime

¹ Any fault that occurs can be considered permanent and therefore further use of the faulty resource can be prevented. Therefore, diagnosis is not entirely necessary, but it may be helpful in identifying non-permanent faults that can be corrected, allowing further use of the faulty resource.

² Tile independence requires the system to generate and store individual tile information and its corresponding instances. However, no inter-tile information need be generated or stored, thus reducing CAD tool effort and storage requirements.

problems on-line, minimizing the amount of system downtime and eliminating the need for outside intervention. The flexibility that this approach provides allows for application-specific solutions. The degree of fault-tolerance can be adjusted to meet timing constraints, resource limitations, or estimated logic block and interconnect reliability.

2 Example

Consider the Boolean function $Y=(A \wedge B) \wedge (C \vee D)$, which might be implemented in a tile containing four logic blocks as shown in the Figure 1. This partitioning contains one spare logic block, which is available if a fault should be detected in one of the occupied logic blocks. Upon detecting such a fault, an alternate configuration of the tile is activated which does not rely on the faulty logic block. Each implementation is interchangeable with the original, as the interface between the tile and the surrounding areas of the design is fixed and the individual configurations implement the same function. The timing of the circuit may vary, however, due to the changes in routing.

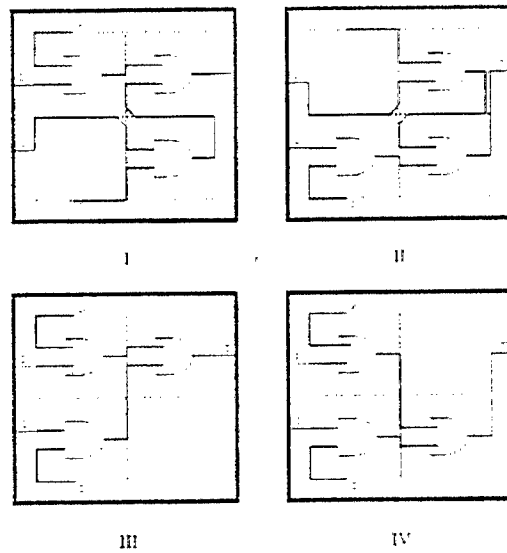


Figure 1: Tolerating logic faults

Tolerating local interconnect faults can be handled in much the same manner. Interconnect can be set aside as unused in each tile configuration³, and when a fault disables an interconnect line an instance not utilizing that line can be activated.

Tolerating faults to global and overlapped segmented interconnect requires a different approach, as much of that interconnect crosses tile boundaries, thus eliminating the independent nature of each tile. Ignoring tile boundaries, interconnect can be set aside that acts as backup for used global and overlapped segmented interconnect. However, the backup interconnect must be able to fulfill the connections of the failed interconnect without requiring an alteration to the affected tiles. Figure 2 shows how a global line (dotted) can act as a backup for several segmented interconnect lines (solid) in the Xilinx XC4000 family. Upon a segmented line sustaining a fault, the backup global line can be activated. This requires only the programming of the proper connections and does not affect the logic in any way. Again, the only difference may be in the timing of the circuit.

³ Throughout the configuration generation for logic reliability, most local interconnect is unused in at least one instance, making it unnecessary to generate many additional instances for local interconnect reliability.

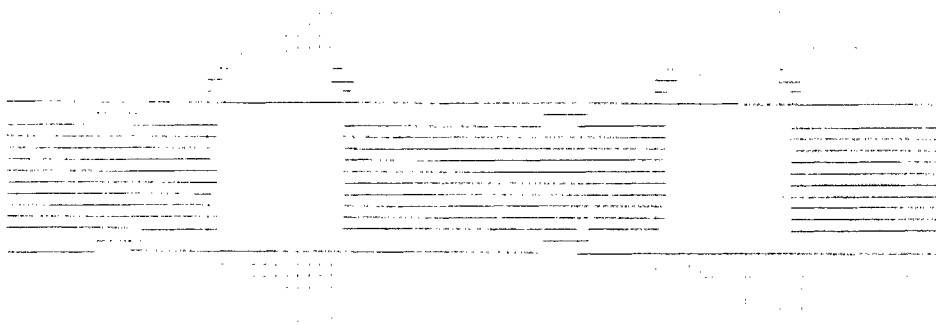


Figure 2: Tolerating inter-tile interconnect faults

2.1 Motivation and Fault-Models

There are two major sources of logic faults in FPGA systems: cosmic radiation and manufacturing/operating imperfections, both of which can affect logic and interconnect.

Since the size of radiation particles is usually small when compared to the size of a modern FPGA logic block, we selected a cosmic radiation fault model that follows uniform distribution of independent (uncorrelated) failures. Extensive terrestrial efforts to accurately model the rate of such soft faults indicate high variance (several orders of magnitude) depending on factors such as seasonal solar activity, altitude, latitude, device technology, and device materials. Even for the same chip from the same manufacturer, variations by a factor higher than 200 are not uncommon [2]. Experiments indicate that in FPGA-like devices at an altitude of 20 km, error rates significantly higher than once per 1000 hours are common [3]. Also, as circuit devices become smaller, they become more sensitive to soft faults [2]. Radiation error rates in specific current FPGA technologies have also been calculated [4]. If one considers the multiyear life of computing devices and other sources of potential errors (e.g. power surges), the need for fault-tolerance in devices which implement critical functions becomes apparent.

The second class of faults is related to manufacturing imperfections. These defects are not large enough to impact initial testing, but after a longer period of operation they become exposed. Design errors can also cause a device to stop functioning in response to rare sequences of inputs (e.g. due to a power density surge in a small part of design). For this type of model, we follow the gamma-distribution Stapper fault model [5]. The model is applicable on any integrated circuit with regular repetitive structure, including memories and FPGA devices.

3 Interconnect Architectures

The implementation of the algorithm varies depending on the target FPGA architecture. The following sections describe the implementation on three architectures: Sanders' context switching reconfigurable computing (CSRC) technology [6], Xilinx's XC4000 family [7], and Altera's Flex 10k series [8].

3.1 Sanders, CSRC Architecture

The CSRC device is based on new architectural techniques that exploit dynamic reconfiguration via context switching. Each context is identical, and a cross-context sharing mechanism enables data sharing between contexts. The device can switch contexts in a single clock, and non-active contexts can be loaded and configured in the background of active context execution. Each context is composed of logic and interconnect arranged in a hierarchical structure. At the lowest level of logic is the context switching logic cell (CSLC). Sixteen CSLCs comprise one context switching logic array (CSLA), as shown in Figure 3. Each logic cell has a carry-in and carry-out. The sixteen CSLCs are sectioned into four groups that are connected and driven by Level 1 routing.

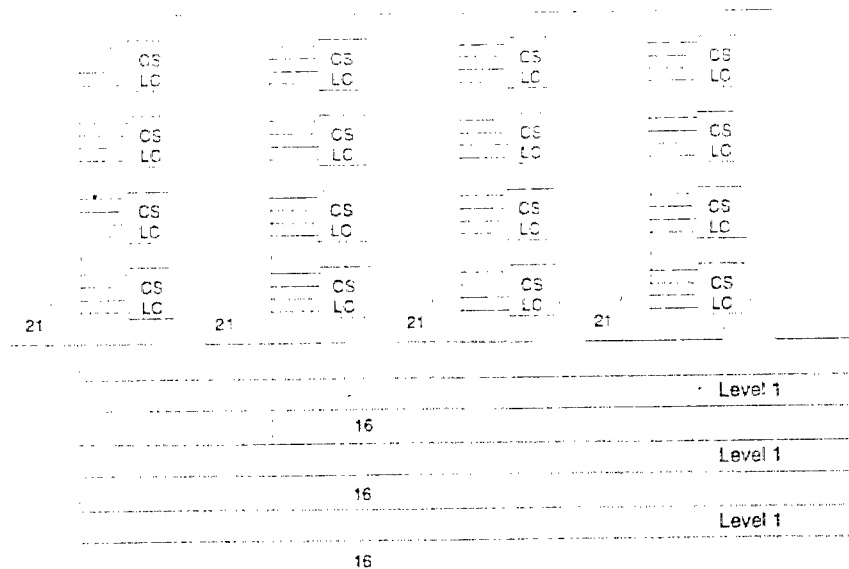


Figure 3: CSLA and Level 1 routing

The next level of logic and interconnect is the 16-bit data pipe which is composed of CSLAs and connected by Level 2 routing as shown in Figure 4. Each context is composed of a set of data pipes that are connected by Level 3 routing, completing the highest level of logic and routing.

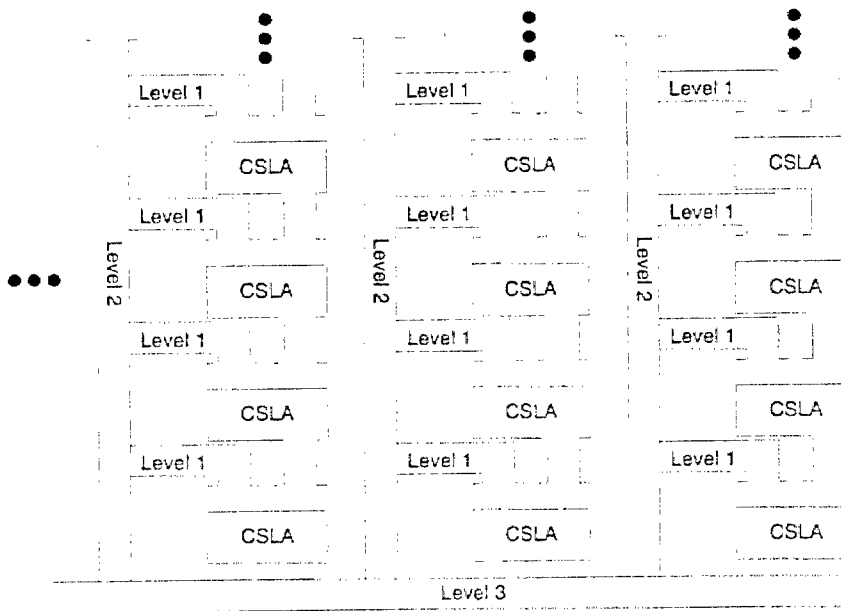


Figure 4: Data pipes and Level 3 routing

Developing a fault-tolerance approach for the CSRC device can be done in two ways: 1) looking at each context as an independent entity, and 2) allowing functionality belonging to one context to be transferred to another in the face of a fault. Analysis reveals that the latter is less desirable. Moving a section of a pipeline, for example, to another context would require switching to the other context in mid-stream before switching back again to complete the context's original function. The CSRC technology is quite capable of performing such a task, but the performance cost would be great. For example, leaving an entire context free to serve as a spare to be programmed and activated in the case of a fault to a portion of an active context is undesirable. In general, such an approach would require a tremendous amount of overhead (X

spare contexts for every Y active contexts⁴) and only be able to tolerate X faults throughout the lifetime of the system. Fault-tolerance in FPGAs can be implemented efficiently because of the inherent redundancy (just as contexts are redundant), but the larger the redundant block, the higher the resource overhead and the smaller number of tolerable faults. Allocating or reserving spare contexts in the CSRC device would be analogous to simply adding redundant FPGAs in a non-CS system.

Looking at each context as an independent entity, a much finer grained approach, helps to alleviate these problems. For example, looking at smaller blocks of redundancy (CSLAs or even CSLCs) creates the opportunity for lower resource overhead and a smaller negative performance impact. CAD tools rarely map logic to maximum density, leaving many CSLAs and CSLCs unused even in a non-fault-tolerant configuration. Using these as redundant blocks eliminates the effective resource overhead. Additional unused resources can, and often should, be added to raise the number of tolerable faults, thus creating resource overhead. Both the naturally unused and additional redundant blocks must also be distributed for easier fault-recovery and a smaller performance impact, but the overhead is still significantly reduced from a larger redundant block approach.

This approach also raises the number of faults that can be tolerated and creates a more efficient and realistic fault model (see Section 2.1). Very rarely would a fault occur that destroys a large portion of the chip. If such a situation arose, it would be unlikely that enough of the chip would remain functional, thus rendering any on-chip fault-tolerance algorithm useless. Most faults that would occur are single faults that affect a small segment of memory (e.g. LUT), logic (e.g. multiplexor or flip-flop), or interconnect (at any level). Such a fault model dictates the use of smaller redundant blocks, as one faulty wire or LUT should not render an entire context faulty.

Breaking the CSRC device contexts into independent fault-tolerant blocks (tiling) can be done in a number of ways. Selecting the most efficient can be application dependent. The pipeline nature of the interconnect makes it difficult to have a single CSLC be redundant for the others in its array (see Figure 3). If a logic cell in the middle of a 4-cell pipe portion should fail, it may not be possible for the CAD tool to route the proper signals to reach the redundant cell. Therefore, it becomes necessary to look at a slightly larger block, i.e. the 4-cell pipe portion. Each portion has the same connectivity, making it possible for one pipe portion to be redundant for any other in the array. Thus, each array has one 4-cell pipe portion that is redundant for the other three, and three other configurations are generated that would be instantiated upon a failure to one of the active pipe portions.

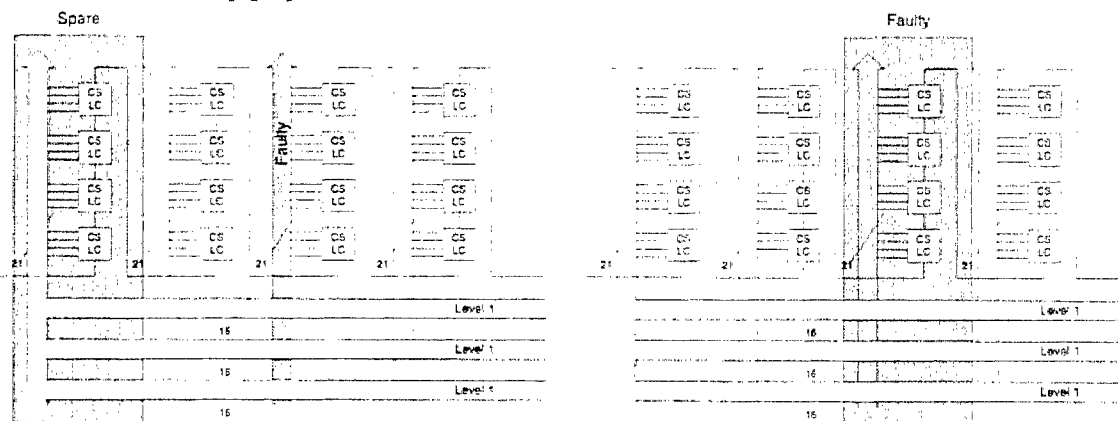


Figure 5: Original and recovered configuration after an internal CSLA fault

Almost all types of failures can be tolerated at this level. Faults to the cells, the cell pins, and the wires leading from the Level 1 routing to the cell pins can all be tolerated, as each can be

⁴ Sanders' CSRC device has four contexts. With three active contexts and one spare, resource overhead would be 25% while being capable of tolerating only one fault.

attributed to a specific pipe portion. Simply switching to an array configuration not utilizing the pipe portion containing the faulty hardware tolerates the fault. Figure 5 shows how a CSLA may be reconfigured if a fault were to occur in the routing from Level 1 to the logic cell pin wires in pipe portion three (or anywhere within the third pipe portion).

One problem with this tiling approach concerns inter-pipe interconnect. This interconnect includes the Level 1 routing and the carry lines, as such lines cannot be attributed to a single pipe portion. The connections among Level 1 routing are plentiful and flexible enough that the CAD tools can find acceptable routing if a Level 1 line should fail, but configurations must be ready at runtime that do not necessitate in the field CAD use. Therefore, configurations must be generated at design-time that do not make use of each Level 1 line in at least one configuration, much in the same way that each pipe-portion is unused in at least one configuration.

Tolerating faults to the carry line is more difficult, as such a fault potentially renders two pipe portions inoperative. The backend CAD tool deals with the carry lines when there is a break in the pipe (i.e. a middle pipe portion is faulty and, therefore, unused) during its design-time configuration generation. However, if the carry-out of one pipe portion and the carry-in of another are faulty, an inter-pipe portion problem arises. The CAD tool may be able to implement the array's functionality without using the carry line (i.e. each carry line unused in at least one configuration). If not, two pipe portions may have to be set aside for the array, or such a fault may be considered intolerable within the array⁵.

This problem can be resolved through a second tiling approach that involves looking at entire CSLAs as the redundant block size. This can be achieved on the CSRC device by using the exact same steps as the previous approach but simply applying them to larger areas: 4-cell pipe portions become CSLAs, Level 1 routing becomes Level 2, etc. Configurations can be generated that leave one CSLA to be redundant for the active arrays in its 16-bit data pipe (see Figure 4). The problems that existed for the first approach exist again at this next level, and they can be dealt with in the same manner. Level 2 interconnect can be reserved as unused in various contexts, just as Level 1 lines were, and the carry lines in and out of the arrays pose the same problem as before and must be dealt with similarly. But, this approach solves the carry line problem at the logic cell level. If a fault occurs in a logic cell carry line, the entire array can be disabled as faulty. This approach also is more efficient if there are highly correlated faults (see Section 2.1) which may render entire arrays faulty.

The approach can also be taken up to the next level with pipes of CSLAs being the redundant block size with the Level 3 routing and the carry lines in and out of the pipe posing the same problems. But again, redundancy at this next level solves the carry line problem from the previous level and tolerates correlated faults that may disable entire pipes.

Choosing the proper level of redundancy may depend on many factors, including the application, the desired level of fault-tolerance (the number and types of faults to be tolerated), the amount of spare resources, and the acceptable amount of overhead (timing and area). Often times, the best technique will be a fractal-like approach combining all of the above levels. Within each logic array, there can be a redundant 4-cell pipe portion. Within each data pipe, there can be a redundant CSLA, and within each context, there can be a redundant data pipe (see Figure 6). It can even be taken to the extreme addressed above in which each device contains a redundant context⁶. Although such an extreme may not be efficient to implement or even be practical within the given fault model, a lower level fractal-like approach may be most desirable for many reasons. First, the CSRC device is inherently self-similar and hierarchical in design and is

⁵ Considering such a fault intolerable under the given approach is a reasonable concession. The four carry lines occupy a relatively small area and, therefore, are less susceptible to faults than the other interconnect or logic which occupy a much larger area of the die.

⁶ A problem does arise when considering a level between redundant data pipes and redundant contexts, as such a level would require tolerating faults to the cross-context data sharing mechanism. The current proposed approach does not tolerate faults to this mechanism.

therefore well suited for such an approach. Second, it emphasizes the flexible nature of the general approach. That is, different levels of fault-tolerance and various acceptable amounts of overhead can be achieved for specific applications within the same general algorithm. Third, each consecutive level solves problems that arise at the previous (e.g. faults to the carry lines). Finally, a wider array of fault models can be accommodated. Isolated single faults can be handled within individual arrays, as there is no need to render large sections of the chip useless for a small fault. Conversely, the higher levels can more efficiently tolerate a large number of correlated faults that may render entire arrays or even entire data pipes inoperable.

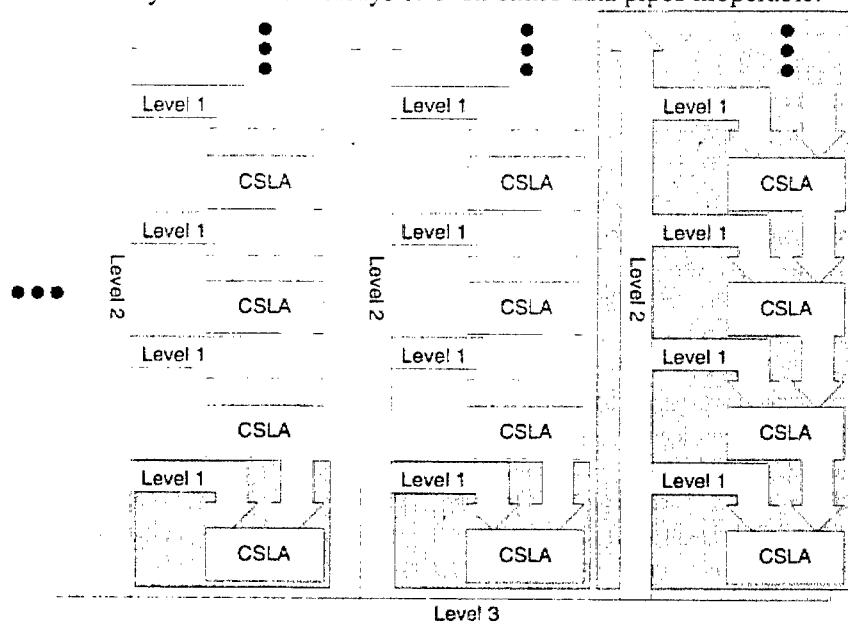


Figure 6: Hierarchical redundancy (shaded areas are spares)

The CSRC architecture is well suited for the proposed fault-tolerance approach. Tiling lines can be drawn at many levels, enhancing the approach's flexibility and thereby increasing the level of fault-tolerance (both number and types of faults) while decreasing area and timing overhead.

3.2 Xilinx, XC4000

Implementing the same algorithm on the Xilinx XC4000 family requires many alterations, but the general algorithm remains unchanged. The main architectural features that require the alterations are the non-fractal and non-hierarchical nature of the family and the wide use of segmented, overlapping interconnect. The former requires that the tiling algorithm be altered, and the latter requires the use of the inter-tile interconnect approach described in Section 2.

The logic in the 4000 family is not broken up into cells, arrays, and pipes as the CSRC device. Instead, there is simply a general array of configurable logic blocks (CLBs) which, along with the local interconnect, can be tiled into smaller groups of CLBs. Figure 7 shows an example of a small design (from the PREP benchmark set) implemented on the 4000 architecture. The first shows the original layout, and the second shows the design after tiling and with one configuration for one tile identified with two spare CLBs.

The placement and shape of the tiles are determined by the following three key factors listed in decreasing order of importance: amount of interconnect across the tile interface, tile logic density, and tile size. Tiling lines are drawn across areas with little inter-tile interconnect to ease the interface locking process and minimize the performance degradation. The logic density of each tile must allow some unused logic for redundancy and should be flexible and malleable to enable various configuration possibilities. Tile size is also a factor, as large tiles may incur large overhead and low fault-tolerance levels as described in Section 3.1. If the first tiling attempt does not meet the user area or fault-tolerant specifications, the algorithm must repeat and find a

clearly supports the approach, but such connections were thought to be unnecessary in the specific 4000 family implementation. The only cost of adding such connections would be additional configuration bits, making the eventual implementation of the algorithm on the architecture a possibility.

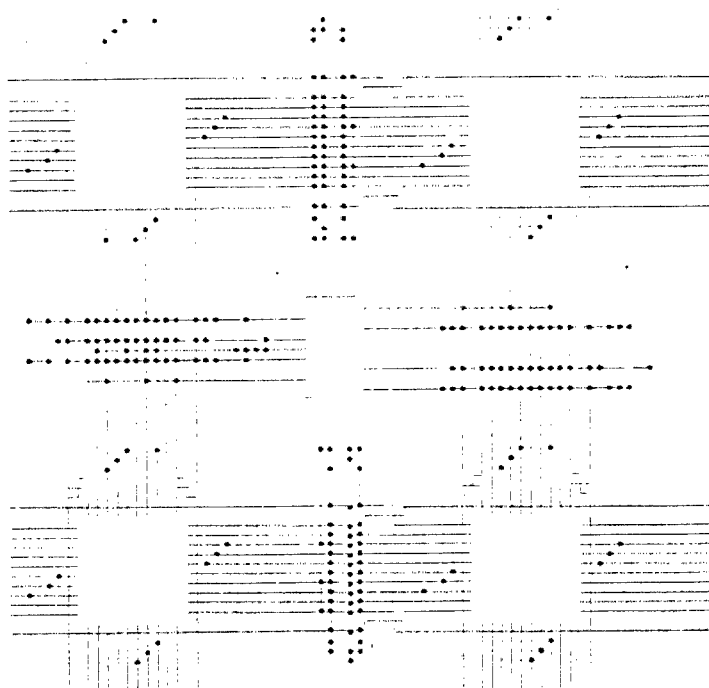


Figure 9: Sample of interconnect connections available on XC4000 family

The one feature of the Xilinx family that eases algorithm implementation involves the absence of inherent carry lines. This benefit is outweighed, however, by the inter-tile interconnect dilemma.

3.3 Altera, Flex 10k

The Altera Flex 10k family is more similar to the CSRC architecture than the Xilinx XC4000 family. The hierarchical structure returns, and the interconnect is more contained. Therefore, the implementation on this architecture is similar to that on the CSRC device.

The unit analogous to the CSLC is the logic element (LE), eight of which comprise the logic for a logic array block (LAB), which is analogous to the CSLA. The local interconnect within a LAB is also quite contained, allowing for a redundancy similar to that used in the CSLA, as it is structured like the Level 1 interconnect. One LE can be redundant for the others within the group of eight, and local interconnect can be set aside in each instance of the LAB generated at design-time by the CAD tools. Figure 10 shows the internal structure of a LAB and how the block can recover from an LE (or associated interconnect) fault.

The hierarchical structure that makes for an efficient implementation on the Flex 10k family as was done for the CSRC device also creates the same problems that existed for the CSRC device. LEs have carry lines that cannot easily be made fault-tolerant within a single LAB. Fortunately, Flex 10k similarities to the CSRC device allow for similar solutions to be available. Therefore, the next level of redundant block size must be inspected, beginning the fractal-type approach for this architecture.

Groups of LABs form logic arrays, similar to the pipe level of the CSRC device, and the row or column⁷ interconnect is analogous to the Level 2 routing. One LAB can be redundant for the

⁷ Row or column interconnect can both be used for the analogy, depending on the direction the pipe is laid. Row interconnect is appropriate if the pipes flow horizontally. In such a case, the column interconnect then becomes analogous to Level 3 routing on the CSRC.

different tile partition. Once tile lines are drawn, the independent tile implementation becomes quite similar to that for the CSRC device. Instances of each tile are generated at design-time that leave a portion of the tile (CLBs and interconnect) unused. When a fault occurs, a tile instance can be activated that does not utilize the faulty resource.

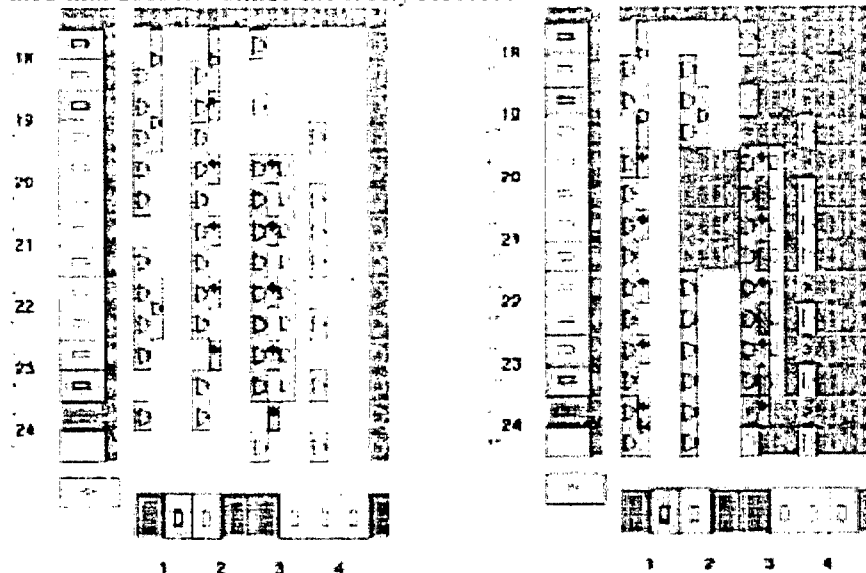


Figure 7: PREP 5 before and after tiling with one tile configuration identified

The second change for implementation on the 4000 family involves the inter-tile interconnect. The 4000 architecture contains many lines that cross tile boundaries, and many are segmented and overlapped. Figure 8 shows an example of how the use of the inter-tile interconnect algorithm from Section 2 could be implemented on the 4000 architecture. The solid lines represent the segmented interconnect, and the dotted are the global lines. The bold line shows the signal before and after a fault is detected on one of the segmented lines and the signal switches to a backup global line.

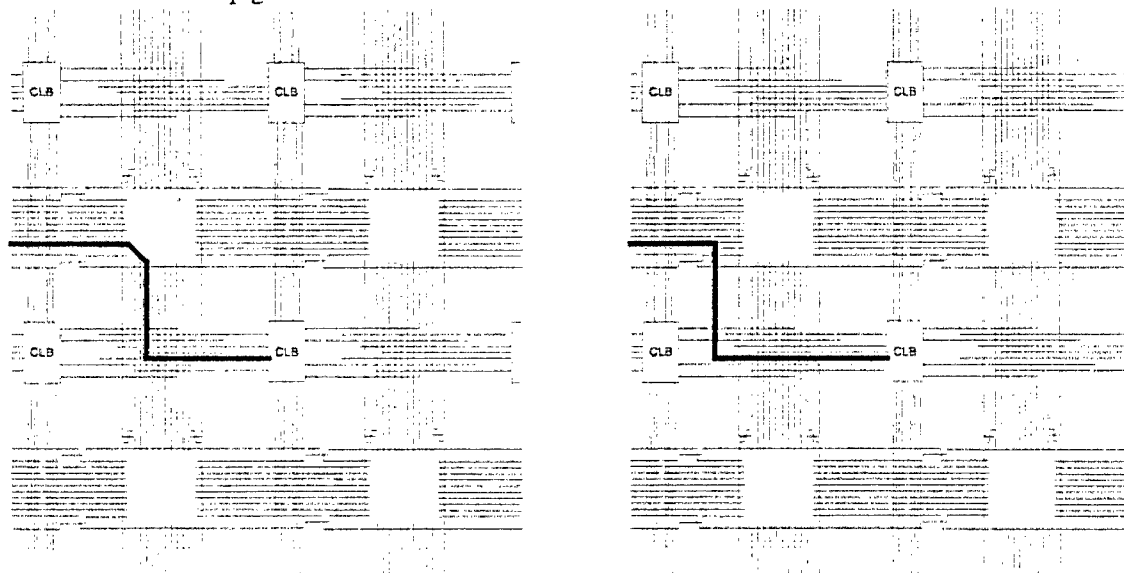


Figure 8: Tolerating inter-tile interconnect faults on the Xilinx XC4000 architecture

The solution works theoretically on this architecture, but the current implementations would not allow the algorithm to work in practice. Although lines could be made available to backup inter-tile lines, the connections do not currently exist for such an implementation, preventing the signals from making the proper routing alterations (see Figure 9). The general architecture

others within its pipe, and configurations can be generated for each LAB that may absorb a fault. Again, carry lines may flow in and out of pipes, potentially requiring that entire logic arrays also have a spare on the device, in the same way that pipes could have a spare on the CSRC device.

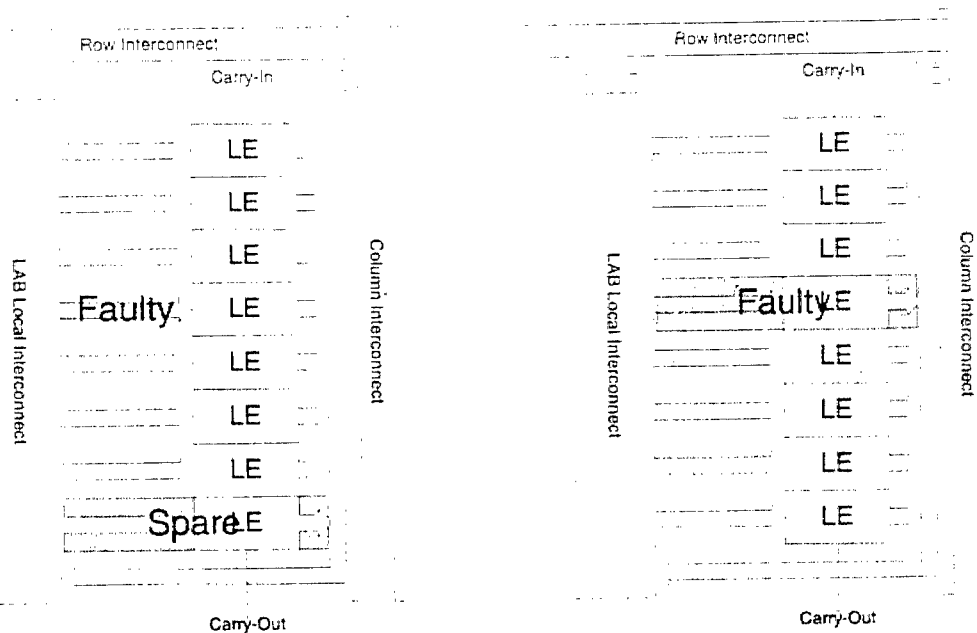


Figure 10: Original and recovered configuration after an internal LAB fault

Using these levels of redundant blocks in combination on the Flex 10k architecture can produce the same benefits as those described for the CSRC architecture, including reduced area and timing overhead and increase levels of fault-tolerance. Figure 11 reveals the potential hierarchical structure of redundant blocks for the Flex 10k architecture.

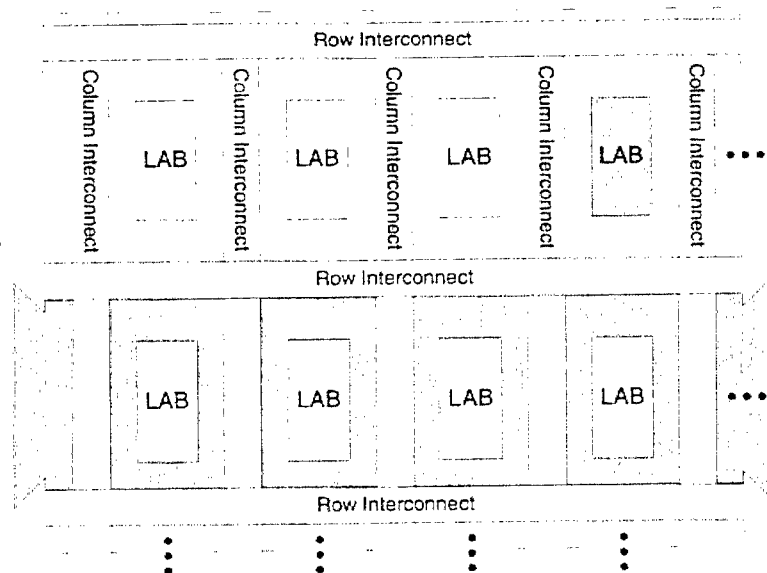


Figure 11: Hierarchical redundancy (shaded areas are spares)

4 Conclusions

Efficiently implementing a runtime fault-tolerance algorithm on a variety of architectures shows the flexible nature of the algorithm and reveals architectural features that enhance or hinder the approach. For all architectures, area and time overhead remain low, as does required

memory and CAD tool effort. Runtime implementation of the algorithm on each architecture is also straightforward and minimizes system downtime. The hierarchical nature and minimal segmented overlapped interconnect of Sanders' CSRC architecture and Altera's Flex 10k family create the greatest opportunity for efficient implementation.

5 Acknowledgements

This work was supported by the Defense Advanced Research Projects Agency of the United States of America, under contract DAB763-95-C-0102 and subcontract QS5200 from Sanders, a Lockheed Martin company.

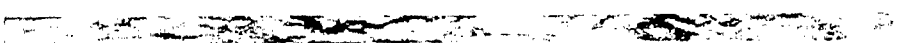
6 References

- [1] J. Lach, W.H. Mangione-Smith, M. Potkonjak, "Low Overhead Fault-Tolerant FPGA Systems", *IEEE Transactions on VLSI Systems*, vol. 6, no. 2, pp. 212-221, 1998.
- [2] J.F. Ziegler *et al.*, "IBM Experiments in Soft Fails in Computer Electronics (1978-1994)", *IBM Journal of Research and Development*, vol. 40, no.1, pp. 3-18, 1996.
- [3] T.J. O'Gorman *et al.*, "Field Testing for Cosmic Soft-Error Rate", *IBM Journal of Research and Development*, vol. 40, no. 1, pp. 51-72, 1996.
- [4] R. Katz *et al.*, "Radiation Effects on Current Field Programmable Technologies", *IEEE NSREC/Transactions on Nuclear Science*, vol. 44, no. 6, pt. 1, pp. 1945-1956, 1997.
- [5] C. H. Stapper, "A New Statistical Approach for Fault-Tolerant VLSI Systems", *The Twenty Second International Symposium on Fault-Tolerant Computing*, pp. 356-365, 1992.
- [6] S.M. Scalera, J.R. Vázquez, "The Design and Implementation of a Context Switching FPGA", *6th IEEE Symposium on FPGA-Based Custom Computing Machines*, 1998.
- [7] Xilinx, *The Programmable Logic Data Book*, San Jose, CA, 1996.
- [8] Altera, *Data Book*, San Jose, CA, 1996.



Dynamic Circuit Generation for Solving Specific Problem Instances of Boolean Satisfiability

Azra Rashid, Jason Leonard, William Mangione-Smith
Electrical Engineering Department
University of California, Los Angeles
Los Angeles, CA 90095-1594



Dynamic Circuit Generation for Solving Specific Problem Instances of Boolean Satisfiability

Azra Rashid, Jason Leonard and William H. Mangione-Smith

The Department of Electrical Engineering
The University of California
Los Angeles, CA 90095-1594
{azra, leonard, billms}@icsl.ucla.edu

Abstract – Optimization and query problems provide the best clear opportunity for configurable computing systems to achieve a significant performance advantage over ASICs. Programmable hardware can be optimized to solve a specific problem instance that only needs to be solved once, and the circuit can be thrown away after its single execution. This paper investigates the applicability of this technology to solving a specific query problem, known as Boolean Satisfiability. We provide a system for capturing the complete execution cost of this approach, by accounting for CAD tool execution time. The key to this approach is to circumvent the standard CAD tools and directly generate circuits at runtime. A set of example circuits is presented as part of the system evaluation, and a complete implementation on the Xilinx XC6216 FPGA is presented.

1 Introduction

One of the most interesting questions facing configurable computing researchers is: When does a configurable computing system exceed the performance of both software and custom hardware? For many people the answer to this question appears to be obvious: never. ASICs will always have a faster clock rate than programmable hardware and can be customized to the particular application. Thus, they reason that dedicated hardware will always achieve higher performance. We believe that programmable hardware *can* be used in a configurable computing system to exceed the performance of alternative ASIC and software solutions. The key to achieving this goal is to exploit opportunities to customize the programmable hardware to a specific problem instance rather than the general problem. For example, integer division is difficult to implement in high-speed digital circuits, at least when compared to addition, subtraction and multiplication. However, a circuit that divides by a specific number is smaller and faster, regardless of what that number is. The key to beating the performance of an ASIC is to use a circuit that is specialized to both the application and a particular problem instance, i.e. data set.

We prefer to refer to such a device as a problem-specific integrated circuit, or a PSIC.

Specializing hardware to data certainly is not a new concept to configurable computing, rather it is one of the core characteristics that has been well investigated. PSICs are a particularly aggressive form of data specialization in that they incorporate the total data set. Without any runtime input, the PSIC always produces the same result. Thus, any specific PSIC is run once, its result is noted, and then the circuit is thrown away. Such an approach can be used with ASIC technology, but is not economically viable.

Three basic properties are essential for a particular application to be well suited to PSIC technology:

1. The application, without specialization, should be well suited to hardware implementation. If the fundamental data-paths are poorly supported by existing FPGA devices, it is much less likely that a specialized version will be implemented efficiently, though exceptions likely exist.
2. The application must present opportunities for data-specific optimization that can be identified efficiently. In particular, the optimizations must result in structural changes to the circuit, either through affecting the number of components or their connectivity. Without structural variation, a registered ASIC approach could be a viable alternative.
3. It must be possible to efficiently implement the optimizations. If execution time is simply transferred from hardware execution to the CAD tool the total system performance will not really increase.

We believe that a multi-stage development process is needed to help keep such a large effort focused.

1. *Study the Application.* The best applications to work with are those that have some existing algorithm base, preferably on fine-grained parallel processors.

2. **Identify Structured Opportunities for Data-Specific Partial Evaluation.** Many applications present little useful opportunities for data-specialization, although they may map well to FPGA hardware acceleration. These applications must be avoided.
3. **Develop Components for Dynamic Instantiation.** A number of researchers in the CAD and configurable computing communities talk of "parameterized macros". The idea here is that there is some structure to many families of components, such as adders. A system should know how to construct many adders, whether they are 3-bit or 12-bit. The difficulty in using parameterized macros with dynamic circuit generation is that all of the information for physical placement must be specified in a manner that is easy to access and manage in order to reduce runtime overhead. For these reasons, our initial attention will be on the Xilinx XC6200 series FPGA, the only available open FPGA architecture.
4. **Break the Design into Fixed and Variable Components.** Most of the existing configurable computing systems have required a harness for managing data as well as I/O. These structures should be designed once and reused.
5. **Manually Design Multiple Configurations.** We need to know that hand-optimized configurations work well before designing the dynamic control software. If we cannot manually implement the optimizations for a properly sized problem instance, then it is unlikely that we can design software to manage the process.
6. **Design Control Software.** This software will run on the processor part of the configurable computing system and manage dynamic optimization. Based on runtime events, such as specific queries from a user interface or partial solutions, the software will invoke the application-specific optimization algorithms. These algorithms implement the process conducted by hand in step 5: instantiate the proper macros into the existing control harness and then restart execution.

This paper reports on our progress through step 5. An initial design of the control software is in place, but all current examples are based on manual development.

1.1 Boolean Satisfiability

Our initial efforts in developing the PSIC technology have focused on the problem of Boolean Satisfiability. The problem of Boolean Satisfiability (SAT) can be phrased in a number of ways. The most common form is from Garey and Johnson [1]:

Given: Variable set U , a subset B of the set of 16 possible Boolean connectives, and a well-formed

Boolean expression E over U and B . Question: Is there a truth assignment for U that satisfies E ?

An equivalent formulation of this problem is as a multi-level logic circuit. Such a circuit can be translated efficiently into conjunctive normal form¹ (CNF), and any Boolean function in CNF form can be directly implemented as a logic circuit without any translation. The benefit of working with an arbitrary combinatorial circuit is that SAT can be invaluable in developing logic test patterns [2] and other operations on arbitrary combinatorial hardware without first translating the form. Boolean satisfiability is known to be NP-complete [3]; this project attempts to reduce the runtime by a constant term, not reduce the fundamental complexity.

The classic technique for solving SAT problems in CNF involves the Davis-Putnam algorithm [4]. An unbound variable is set one way, and the function is analyzed to determine the implication on all other inputs. For example, if the underlying function is a two-input XOR, when one variable is assigned a value it immediately implies that the other input must have the complementary value. If a value assignment results in a logical contradiction, i.e. a previously assigned variable has a conflicting implied value, then the new assignment must be rejected. Variables are assigned in a specific order, and backtracking is used to explore the entire search space.

An alternate approach answering the SAT problem has been proposed by Abramovici and Saab [5], and is based on the PODEM test generation algorithm developed by Goel [6]. This approach constructs a *forward* and *backward* network. The forward network propagates primary input assignments forward through the original combinatorial circuit. The backward network propagates a primary output's objective (1 or 0) along a single path to a primary input whose value is currently X (don't care). This iterative process continues, making one primary input assignment per iteration, until the output objective is satisfied, or the algorithm is unable to determine a suitable primary input vector assignment for the desired output value. In the case of a logical conflict, a *backtracking* mechanism restores the state of the network prior to the last primary input assignment and assigns a complementary value to that primary input.

¹ A logical formula consisting of a conjunction of terms, where each term is a disjunction and no disjunction contains a conjunction. Such a formula might also be described as a product of sums.

2 Related Work

The relevant previous work can be divided into two main areas, the first using hardware description languages to implement PSIC solutions, and the second involving dynamic circuit generation.

Babb et al. made one of the earliest attempts to experiment with the PSIC approach [7]. This work translated graph problems to digital circuits, and then followed the existing CAD tool flow to map these circuits onto FPGA technology.

Recently, a number of researchers have investigated the benefits of mapping SAT problems to FPGAs [5, 8-12]. Four different algorithms were implemented, with each system reporting significant speedups. However, none of the reports measured the CAD tool execution time, i.e. the time it takes to synthesize and place-and-route individual circuits. Zhong et al. mentioned that these tools could require hours to complete, but ignored this factor when comparing their system performance to the state-of-the-art software system. Shand worked on a closely related problem [13], though his hardware was not highly customized to the specific data set. In developing his configurable computing application, Shand discovered a number of novel tricks that were equally applicable to his software implementation, a common occurrence that is often ignored in the field.

Efforts to apply partial evaluation and dynamic circuit generation to programmable hardware have gained momentum in recent years. In general, these attempts can be classified in two areas: constant propagation and localized replacement.

Constant propagation as applied to programmable hardware began with the use of constant multipliers in digital filtering. Petersen and Hutchings work is perhaps the most thorough evaluation of constant multiplication applied to FPGAs [14]. One key to achieving high-performance is the use of the small memories inside of many CLB architectures to implement fixed multiplication over a narrow bit width. These lookup tables are specialized to the constant values in order to generate partial products, which are then summed to produce the complete product. The resulting circuits are significantly faster than more general-purpose structures when implemented on FPGAs. Other examples of constant propagation include the modification of ATM switch schedulers to account for virtual circuit priority [15] and DES hardware customized to specific encryption keys [16].

Hutchings has advanced constant propagation further, through localized circuit replacement [17]. His group has invested some effort in the Sandia ATR problem [18]. They were able to design customized pixel circuits image correlation which were approximately 30% smaller than a

corresponding circuit that could be programmed to handle either case. By using the ability to partially reconfigure a National CLAY FPGA, Hutchings was able to "program" in various templates by placing either the on or off pixel circuits as appropriate. We refer to this approach as localized replacement, because while it does involve changing circuit components and not just data values there are no structural changes to the global netlist.

3 Approach

Our hardware implementation approach is very similar to the PODEM mapping employed by Abramovici and Saab [5]. While the PODEM algorithm works with any combinatorial logic circuit, the implementation presented here is constrained to two-input AND/OR gates and inverters. This restriction simplifies the process of hardware macro generation, as the library must only contain two module types, which have essentially the same requirements of physical resources. More complicated circuits can be translated efficiently into this form. An alternate approach would involve developing macro generators that could synthesize N input gates. Such an approach seems feasible, though it has not been pursued.

Each net in the original circuit must be implemented with two physical wires in the forward network, so that the net can carry the necessary ternary values (0/1/X). Furthermore, for each original net there is a pair of backward nets that propagate the objective function. While the number of physical nets has increased by a factor of four over the original circuit, all four physical nets connect the same set of blocks. Thus, while routing demands have increased they remain highly regular and the circuit topology is unchanged. Consequently, there is no increased complexity for either partitioning or routing. These two parallel networks are referred to the forward network, which carries the values of the original circuit, and the backward network, which carries the objective values.

A controller circuit sequentially makes assignments to the original primary inputs, waiting for the forward and backward networks to settle between assignments.

3.1 Functional Blocks

There are two coding schemes, one in the forward network and one in the backward network. The forward network uses "10" to encode logical 0, "01" to encode logical 1, and "11" to encode the third don't care state X. A logical inversion is accomplished by swapping the bits or equivalently, exchanging the two physical wires that carry the ternary value. Thus, inverters are free.

The backward network assigns an *objective flag* and an *objective value* to each net. The objective flag indicates whether or not there is an objective for a net. If the objective flag is 1, the objective value determines its value (0 or 1); otherwise there is no objective for that net. During the execution of the SAT solving circuit, the objectives propagate through the backward network to the primary inputs. Only one primary input's objective flag is set in a given cycle. The controller looks at objective data generated by the backward network along with the outputs generated by the forward network and assigns the appropriate values to the primary inputs. If the primary output becomes 1 (i.e. meets the target objective), the controller raises the *Done* flag and asserts *Satisfied*. When a conflict occurs, a stack maintained by the controller restores the previous cycle's objective flags and uses these values to backtrack. If during the backtracking process the stack becomes empty, the controller asserts the *Done* flag and sets *Satisfied* to 0.

Figure 1. illustrates the logical structure of the AND macro block. The logical structure for the OR macro block is constructed similarly. Rather than using two separate forward and backward modules, each AND/OR macro block integrates the logic of both the forward and backward networks.

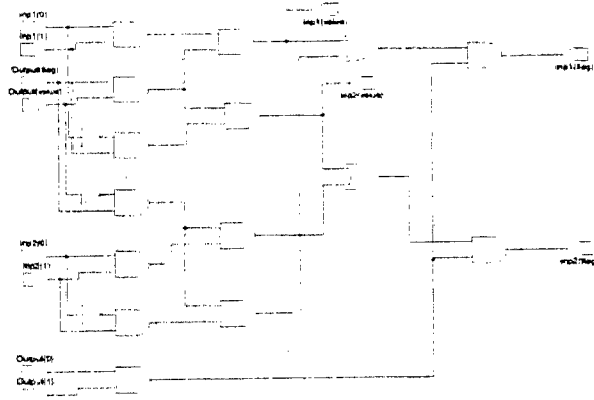


Figure 1: Logic function of AND macro block, illustrating the structure of the forward and backward networks

4 Circuit Generation

Our working assumption for development of all CAD tools is that they must be fast and efficient. At a number of decision points, algorithms were selected that sacrificed some performance and hardware efficiency in the final circuit in exchange for reduced tool runtime. It is our belief that the most important step is to optimize hardware for a specific problem instance and that it is more important to produce a satisfactory solution quickly than to search for a highly optimized circuit.

There are four steps in the CAD tool chain: synthesis, partitioning, placement and routing. A direct form translation is used for synthesis, resulting in $O(n)$ complexity, where n is the number of logic gates. The Kernighan-Lin algorithm is used for partitioning, which results in $O(cn)$ time complexity. Finally, greedy linear time algorithms are used for placement and routing. Each of these algorithms are much more efficient than those typically used in FPGA circuit development because they take advantage of specific information regarding the restricted problem, i.e. satisfiability of combinatorial logic circuits rather than manipulation of unconstrained HDL.

4.1 Synthesis

Circuit synthesis is straight forward, given the structure of the problem. Each circuit is constructed from three components: the controller, the forward network, and the backward network.

The controller block sets the values for primary inputs into the forward network in sequence, and manages backtracking. Thus, a circuit is associated with each of the primary inputs of the original circuit. The controller must also maintain an $n \times n$ stack, where n is the number of primary inputs, in order to manage the backtracking process. As a result, the controller block has size roughly equal to $h(n)v(n)$, where h and v are functions that compute the horizontal and vertical size. However, the controller is only dependant on the number of primary inputs, and not the structure of the original circuit. Thus, it can be immediately instantiated using a macro that is parameterized by the number of primary inputs.

Similarly, each logic gate in the original circuit can be directly replaced by one of two macro blocks. These macros implement the forward and backward networks for the two-input AND and OR.

Synthesis requires traversing the original circuit in a topological order, constructing a new graph with identical topology containing the new function blocks, and instantiating the controller macro after counting the number of primary inputs. These tasks can be completed in linear time.

4.2 Partitioning

The target FPGA will always have a rectangular shape (usually square), as will the controller macro. Thus, if the controller is always located in one of the corners of the chip, two rectangular regions will be left free to hold the logic blocks which implement the forward and backward networks. The partitioning problem involves finding a satisfactory division of the blocks into these

two rectangular regions. For the current implementation, the controller is placed in the lower left corner. The first portion of the forward and backward networks is mapped in the lower right region with the forward network moving left to right, and the remainder of the circuit is mapped in the upper region with the forward network moving right to left. This floorplan is illustrated in Figure 2.

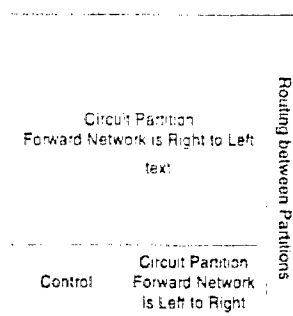


Figure 2: Floorplan for final layout

We use a modified form of the Kernighan-Lin approach for graph partitioning [19]. Since the circuit is a DAG, there is a critical path, which can be determined in linear time. Blocks on the critical path are allocated to the partitions in proportion to the width of the physical partitions; i.e. in this case, roughly 60% of the critical path would map into the upper partition. The blocks in the critical path are locked into their partitions. The remaining blocks are allocated in topological order, first into the lower area and then into the upper, so that the total number of nodes in the two partitions is proportional to their areas. Next, the Kernighan-Lin algorithm is invoked to refine the partitions. The weighting function measures the number of hypernets crossing between the two partitions. While better algorithms are known, this approach is recognized as being the best choice for quickly finding a high performance partition in the majority of cases [20].

4.3 Placement

Within each of the two circuit regions, blocks are placed on a grid. This approach is feasible because there are only two types of blocks, and they have the same physical footprint. The mapping grid has enough entries for each of the blocks assigned to the partition, and an aspect ratio (roughly) equal to the physical partition.

Placement is accomplished using the Lin-Du algorithm [21]. The partition inputs and outputs are first placed. Subsequently, the paths are placed in order of criticality, as measured by the length of the net (within the partition). Each block is placed into the free position that results in the lowest total net lengths for the currently placed blocks connected to the input and output hypernets. No

backtracking is done if a component fails to be placed; rather the block is moved to the other partition. If a block cannot be placed in either partition then the entire mapping is aborted to a higher-level recovery routine (e.g. circuit pruning to reduce the search space). This algorithm requires at most two steps for each block in the circuit (one for each partition), and thus executes in linear time.

4.4 Routing

Routing is accomplished by managing a set of routing tracks, which are physical resources that can carry either vertical or horizontal traffic. For example, on the Xilinx XC6200 each of the macro blocks is aligned on a 4-cell pitch. Thus, it is natural to allocate four cells to each routing track, providing routing for one logical net. After logical placement into the virtual grid, routing begins by first routing all vertical nets moving from the left to the right, and next routing all horizontal nets moving from the top to the bottom. A simple greedy compaction is used, where physical nets are allocated to the same track if the track segment is currently free. Again, no backtracking is implemented.

5 Status and Results

The dynamic circuit generation system is being implemented in Java, in order to leverage the JERC system developed by Xilinx [22]. The system reads in a circuit specification and constructs a Java representation where each node is an object. Using this approach, the circuit can be asked to print itself out in one of two forms, either VHDL or a XC6216 configuration. VHDL output is supported in order to debug the high-level system architecture, as well as facilitate comparisons to results from other researchers, who are developing this approach.

Table 1 summarizes the results of applying this tool to a set of combinatorial circuits, and evaluating the approach of using VHDL as an intermediate representation. The first ten circuits were selected from the PREP suite [23], while the final four are test circuits that were developed for use in initial hand routing. Synopsys FPGA Express was used for synthesis, while Xilinx XACT was used for place-and-route. The target selected was the XC4000 family of devices, since this architecture is a better target for automated CAD tools and the software is more mature. The time required for synthesis and place-and-route varied significantly, and there is no clear functional dependence on either the number of primary inputs or the total number of logic gates. However, the compilation time was never less than fourteen minutes, and was more than fifty minutes in two cases. Note that the times in Table 1 do not include the execution times,

since we are not interested in pursuing this path. The circuits are quite small, and a number of software systems (e.g. GRASP [24]) would be able to actually solve the problems in well under a minute.

We initially used hand placement and routing of small circuits in order to better understand the problem, develop the essential component macros, and experiment with heuristics. These circuits are shown in Figure 3 through Figure 6.

Figure 7 shows the physical layout of the AND macro block, corresponding to the logic from Figure 1. While the OR macro block has a slightly different layout it has exactly the same footprint. Our initial hope was to develop a layout with a vertical and horizontal pitch that was a multiple of four cells, given the XC6200 routing structure. Unfortunately, the AND/OR logic appears to require a minimum of 17 cells. Because of this requirement, we have currently adopted a 4x6 layout with a 4x8 placement alignment. This approach allows us to use the remaining eight cells in each macro for patch-up logic that implements fan-in and fan-out from the two input gates.

The final layout for TC2 is shown in Figure 8. A significant amount of resource is dedicated to routing, resulting in relatively low logic density. Currently, the OR and AND macros employ by-4 nets internally. The hand placement avoided routing over these macros. Clearly, a better approach would be to route in one direction, e.g. use the horizontal routes within the macro and the vertical routes over the top of the macros. Another important area for optimization is to better use the ability to split a routing track into multiple segments, and refine the grid placement of each macro in order to reduce the total number of tracks. We are in the process of pursuing both of these approaches.

Table 2 contains the summary data for the three test cases that were hand routed to completion. Test case 5 was not completed due to time limits, though it contains fewer gates and primary inputs than test case 4. Clock periods varied between 195 ns and 340 ns, with over 80% of the delay associated with interconnect. If the approaches discussed above prove effective at increasing the functional density, we suspect that they will also have a significant impact on reducing the delays.

6 Conclusions

This report presents our initial efforts to develop PSIC technology; i.e. an application of configurable computing that may prove immune to performance attacks from ASIC technology. An architecture was presented for implementing arbitrary instances of the Boolean Satisfiability problem on configurable hardware. A set of CAD algorithms was discussed which together form a complete application framework. These algorithms were selected and developed with a bias toward efficient

software runtime, sometimes at the expense of performance for the final circuits. Early results from this tool help to confirm the intuition that VHDL is a poor intermediate language for implementing PSIC applications, specifically because of the time consuming steps of synthesis and place-and-route. We believe that, in order to be well matched to the PSIC approach, applications must exhibit well-structured opportunities for optimization. Configurable computing systems need to exploit these structures in order to avoid executing slow commercial CAD tools, as well as achieve efficient solutions.

Finally, our initial efforts at hand development of SAT solutions have proven useful for developing the structure of the architecture as well as the individual macro blocks. Important insight on routing strategies, which we believe will prove to be the essential problem for all PSIC systems, emerged from this exercise as well.

Acknowledgements

This work was supported by the Defense Advanced Research Projects Agency of the United States of America under contract DAB763-95-C-0102 and subcontract QS5200 from Sanders, a Lockheed Martin company, as well as NSF CAREER award #9734132.

References

- [1] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*: W. H. Freeman and Company, 1979.
- [2] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," *IEEE Transactions on Computer-Aided Design*, vol. 11, pp. 4-15, 1993.
- [3] S. A. Cook, "The Complexity of Theorem-Proving Procedures," *ACM Symposium on Theory of Computing*, 1971.
- [4] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *Journal of the ACM*, vol. 7, pp. 201-215, 1960.
- [5] M. Abramovici and D. Saab, "Satisfiability on Reconfigurable Hardware," 7th International Workshop on Field Programmable Logic and Applications, London, England, 1997.
- [6] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Transactions on Computers*, vol. C30, pp. 215-222, 1981.
- [7] J. Babb, M. Frank, and A. Agarwal, "Solving Graph Problems with Dynamic Computation Structures," *SPIE '96: High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*, 1996.
- [8] P. Zhong, M. Martonosi, P. Asher, and S. Malik, "Accelerating Boolean Satisfiability with Configurable Hardware," *Field-Programmable Custom Computing Machines*, 1998.
- [9] P. Zhong, P. Ashar, S. Malik, and M. Martonosi, "Using Reconfigurable Computing Techniques to Accelerate Problems in the CAD Domain: A Case Study with

- Boolean Satisfiability," Design Automation Conference, 1998.
- [10] J. Cong and J. Peck, "On Acceleration of the Check Tautology Logic Synthesis Algorithm using an FPGA-based Reconfigurable Coprocessor," Field-Programmable Custom Computing Machines, 1997.
- [11] P. Zhong, M. Martonosi, and S. Malik, "Implementing Boolean Satisfiability in Configurable Hardware," International Workshop on Logic Synthesis, 1997.
- [12] T. Suyama, M. Yokoo, and H. Sawada, "Solving Satisfiability Problems on FPGAs," 6th International Workshop on Field-Programmable Logic and Applications, 1996.
- [13] M. Shand, "A Case Study in Algorithm Implementation in Reconfigurable Hardware and Software," Field-Programmable Logic and Applications, London, England, 1997.
- [14] R. J. Petersen and B. L. Hutchings, "An Assessment of the Suitability of FPGA-Based Systems for Use in Digital Signal Processing," in *Field-Programmable Logic and Applications*, W. M. a. W. Luk, Ed. Oxford, England: Springer, 1995, pp. 293--302.
- [15] S. Singh, J. Hogg, and D. McAuley, "Expressing Dynamic Reconfiguration by Partial Evaluation," Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, 1996.
- [16] J. Leonard and W. H. Mangione-Smith, "A Case Study of Partially Evaluated Hardware Circuits: Key-Specific DES," Proc. of Field-Programmable Logic and Applications, London, England, 1997.
- [17] M. J. Wirthlin and B. L. Hutchings, "Improving Functional Density Through Run-Time Constant Propagation," Proc. of Field Programmable Gate Arrays, Monterey, CA, 1997.
- [18] J. Villasenor, B. Schoner, K.-N. Chia, C. Zapata, H. J. Kim, C. Jones, S. Lansing, and B. Mangione-Smith, "Configurable Computing Solutions for Automatic Target Recognition," Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, 1996.
- [19] B. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Systems Technical Journal*, vol. 49, pp. 291-307, 1970.
- [20] M. Sarrafzadeh and C. K. Wong, "The Top-Down Approach: Placement," in *An Introduction to VLSI Physical Design*, McGraw-Hill Series in Computer Science, E. M. Munson, Ed.: McGraw-Hill, 1996.
- [21] I. Lin and D. H. C. Du, "Performance-Driven Constructive Placement," Design Automation Conference, 1990.
- [22] E. Lechner and S. A. Guccione, "The Java Environment for Reconfigurable Computing," Field-Programmable Logic and Applications, London, England, 1997.
- [23] Programmable Electronic Performance Group, "PREP Benchmark Suite #1, Version 1.3," PREP Group, Los Altos, CA 1994.
- [24] J. Silva and K. Sakallah, "GRASP - A New Search Algorithm for Satisfiability," IEEE ACM International Conference on CAD-96, 1996.

Circuit	Function	2-Input Gates	PIs	Synopsys Synthesis	Xilinx Place & Route	Total Time
cm42aA	Logic	27	4	1 min, 55 sec	17 min	18 min, 55 sec
decodA	Decoder	41	5	1 min, 25 sec	13 min	14 min, 25 sec
majority	Voter	14	5	1 min, 31 sec	14 min	15 min, 31 sec
cm138aA	Logic	23	6	1 min, 15 sec	13 min	14 min, 15 sec
cm82aA	Logic	24	5	1 min, 43 sec	17 min	18 min, 43 sec
cm151aA	Logic	37	12	4 min, 27 sec	47 min	51 min, 27 sec
pm1A	Logic	69	16	3 min, 55 sec	21 min	24 min, 55 sec
b1	Logic	17	3	0 min, 39 sec	15 min	15 min, 39 sec
cm163aA	Logic	56	16	6 min, 40 sec	45 min	51 min, 45 sec
cm851	Logic	46	11	4 min, 2 sec	30 min	34 min, 2 sec
TC1	test case	5	3	1 min, 19 sec	13 min	14 min, 19 sec
TC2	test case	11	6	1 min, 53 sec	13 min	14 min, 53 sec
TC4	test case	19	8	1 min, 58 sec	35 min	36 min, 58 sec
TCS	test case	16	7	1 min, 36 sec	15 min	16 min, 36 sec

Table 1: Synthesis and CAD tool execution times for VHDL-based customized SAT circuits



Figure 3: Test case 1

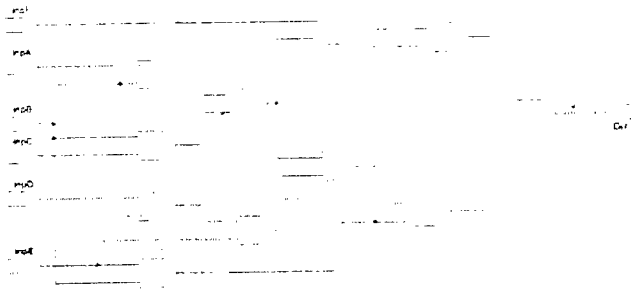


Figure 4: Test case 2



Figure 5: Test case 4



Figure 6: Test case 5

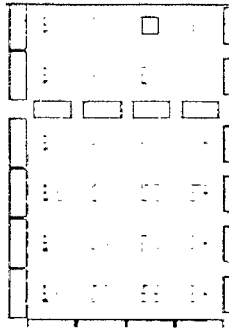


Figure 7: Layout of AND macro block

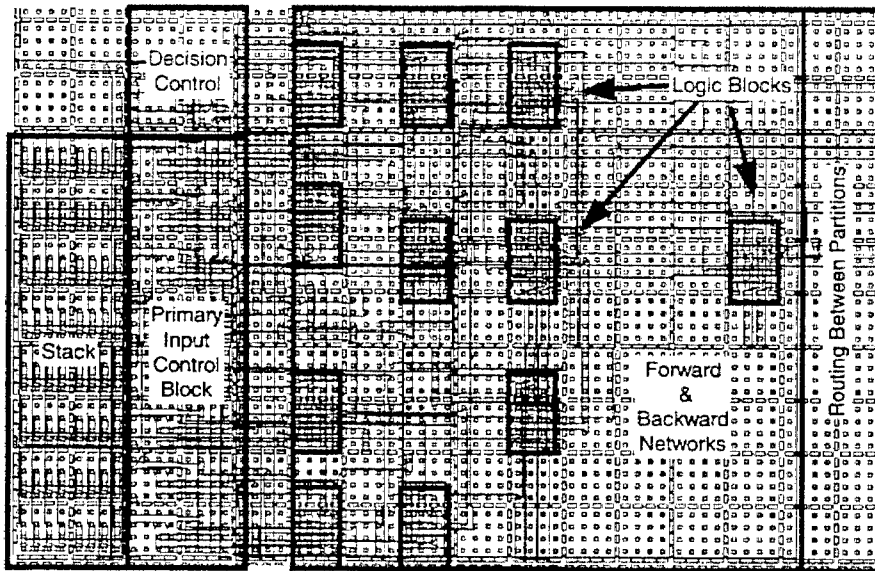


Figure 8: Layout of test circuit #2

Circuit	Clock Period	Cells occupied with logic	Total Area	% of Chip Occupied
TC1	195 ns	522	1.266	31%
TC2	308 ns	680	2.684	66%
TC4	340 ns	895	3.152	77%

Table 2: Summary results from three test circuits

CSRC 1

- Context 1 : 3 Differences in parallel, storage of 16 bit data using 16 three bit Shift Registers. Control to initiate Multiplier context, and store back results in Registers (C, A1, A2, B1, B2, BinSize, AddAddTerm, Child Radius, Alpha0, D).
- Context 2 : 17 X 17 parallel Braun Multiplier (integer & MyFloat), Shifter, and Accumulator. Exponent and mantissa adjustment blocks.
- Context 3 : 2 Adder for parallel addition.
16 bit Registers to store AddAddTerm, SqrtTerm1, AddTerm1
 $B1/(Sqrt(C))$, $D/(Sqrt(C))$, concurrently.
- Context 4 : Square Root Front End.

CSRC 2

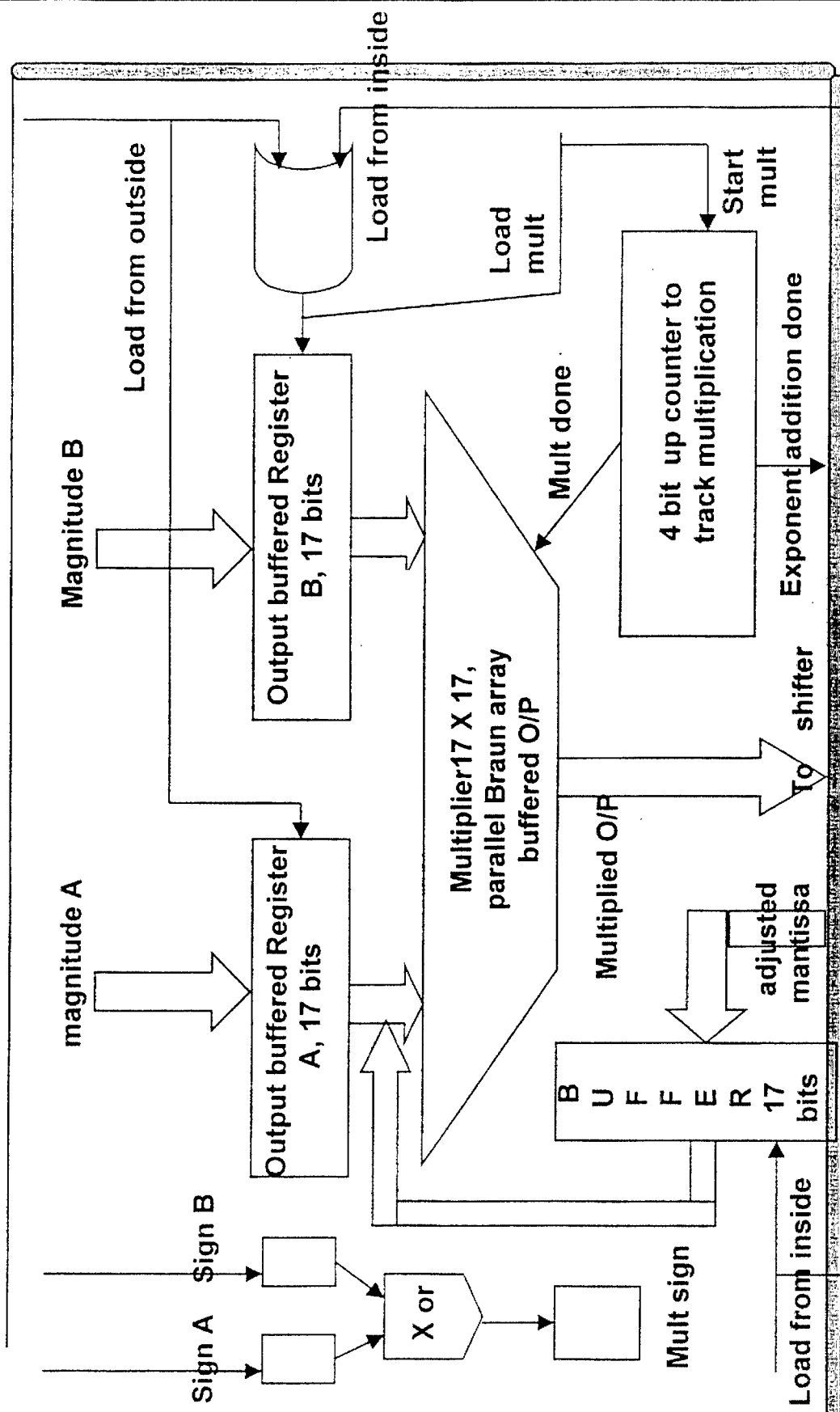
Context 2 : One semi-parallel Divider.

Context 4 : 17 X 17 parallel Braun Multiplier (integer & MyFloat), Shifter, and Accumulator. Exponent and mantissa adjustment blocks

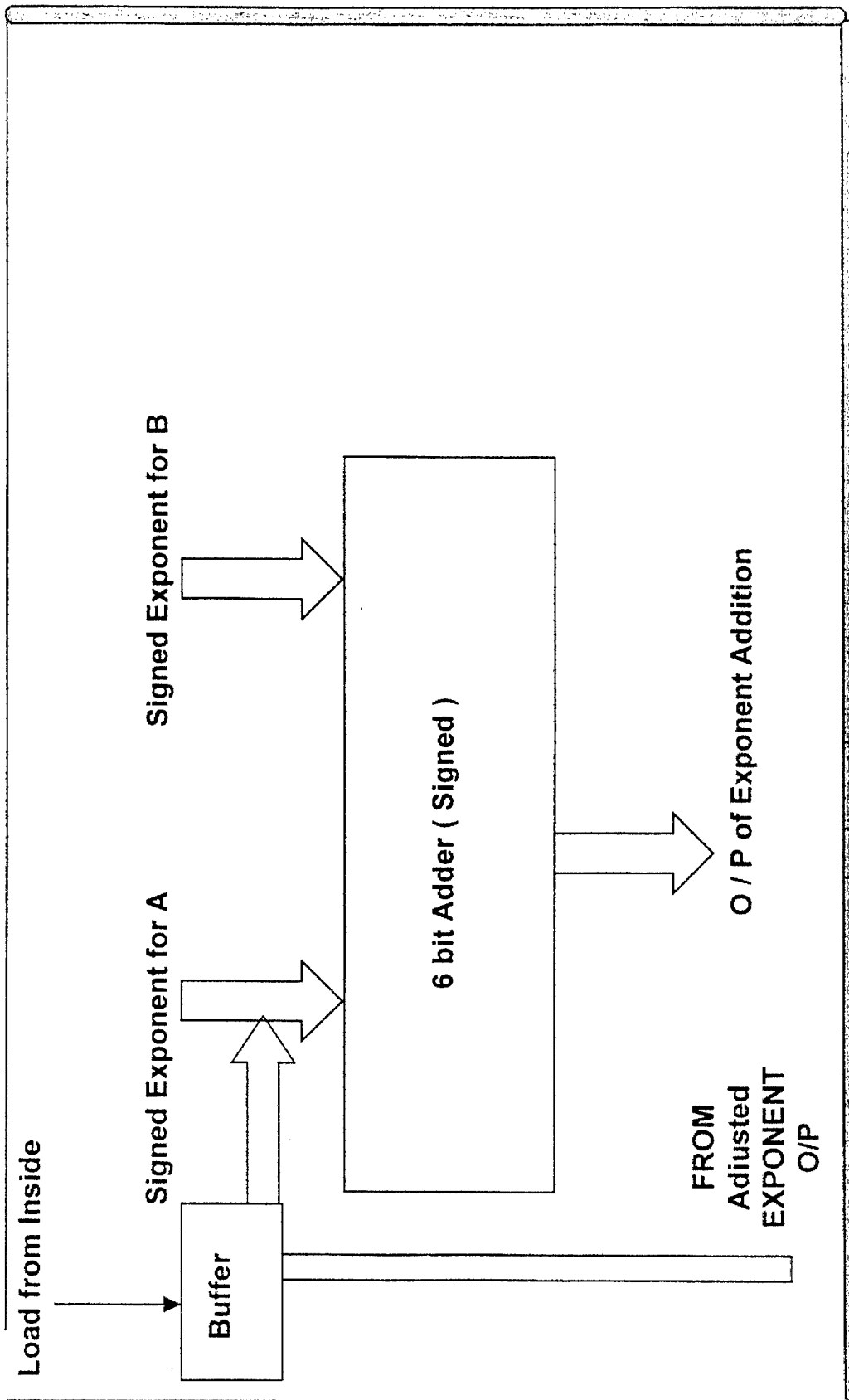
Context 3 : 2 Adder for parallel addition.
16 bit Registers to store AddAddTerm, SqrtTerm2, AddTerm2
 $B2/(Sqrt(C))$, $D/(Sqrt(C))$, concurrently.

Context 1 : Square Root Front End.

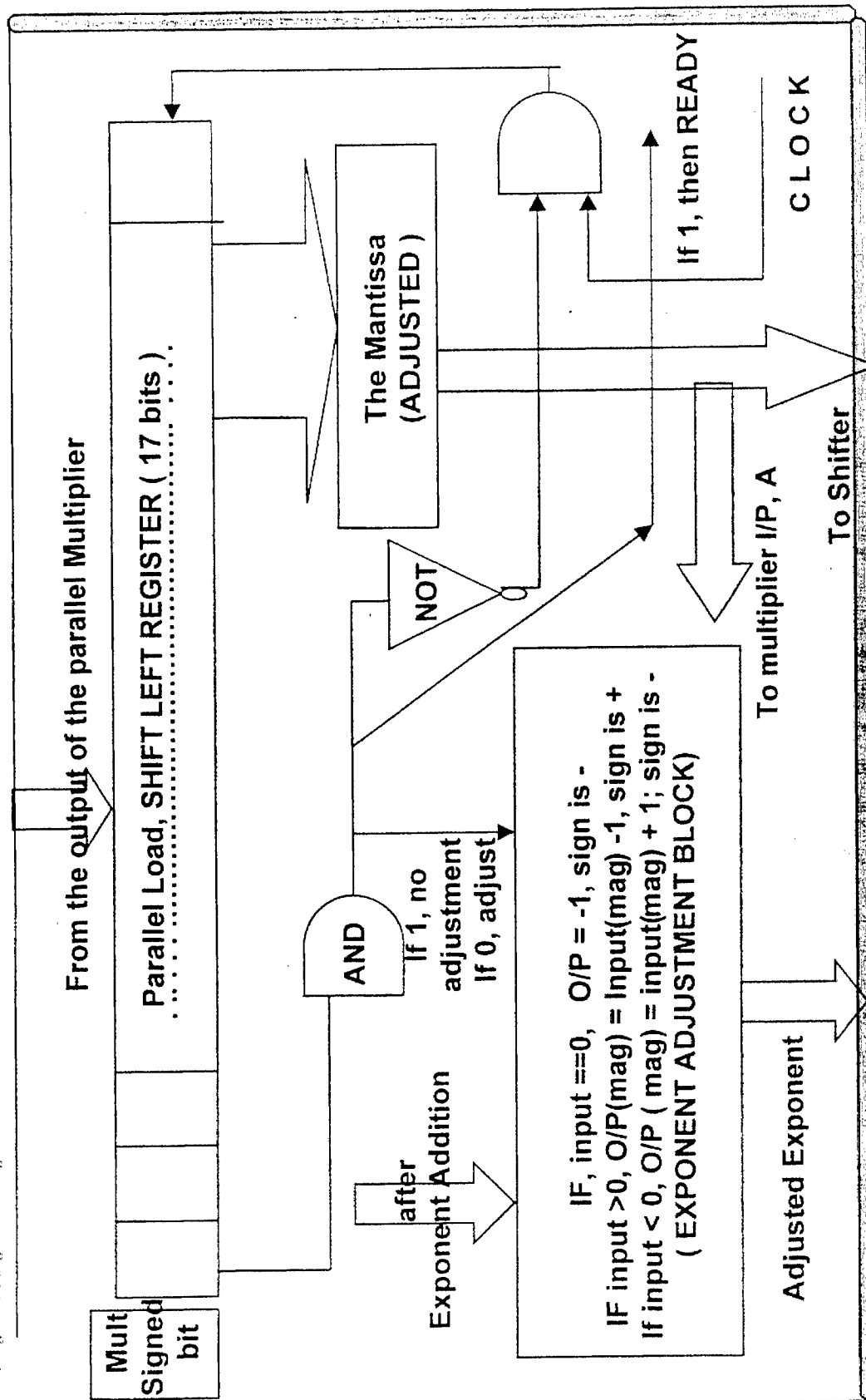
Multiplier (17X17, parallel array) CSRC 1 & 2, context 2 (slide 1)



Exponent Addition due to Multiplication CSRC 1 & 2, context 2 (slide 2)

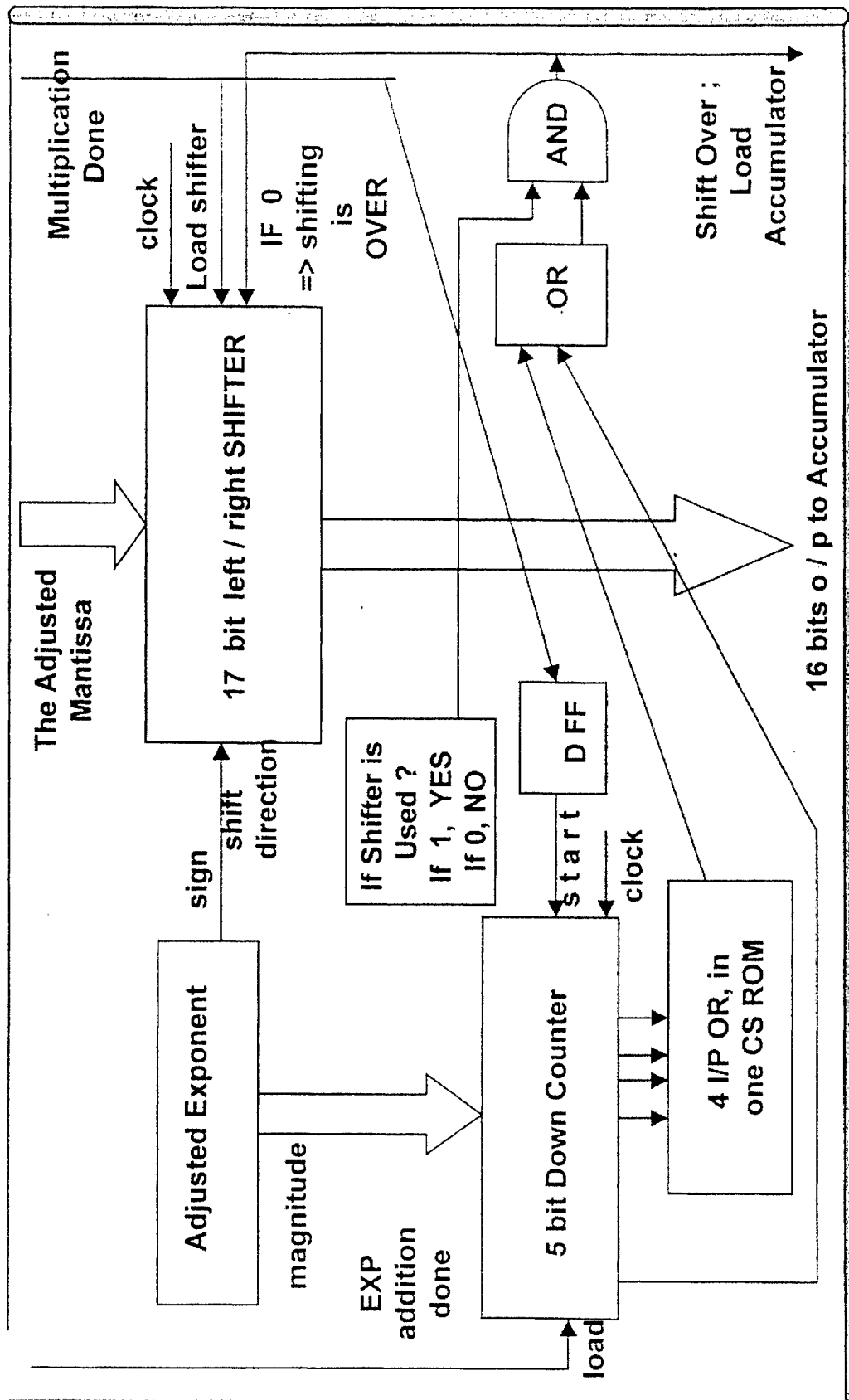


Adjustment after Multiplication CSRC 1 & 2, context 2 (slide 3)

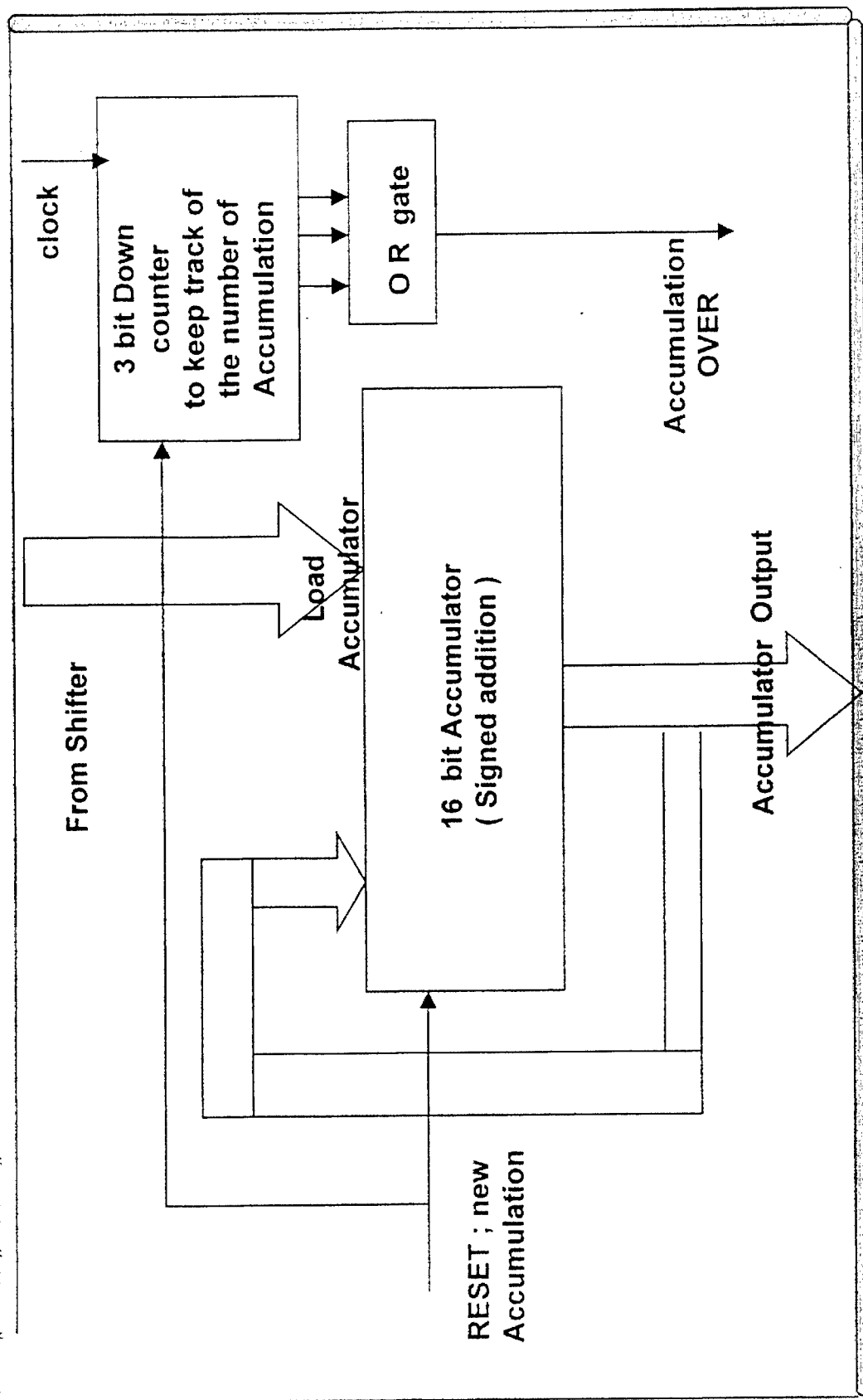


MyFloat to Integer Converter

CSRC 1 & 2, context 2 (slide 4)



Accumulator after Myfloat to Integer Conversion (CSRC 1 & 2 , context 2) (slide 5)



Description of the previous slides

(1) The Multiplier, shifter and Accumulator Context.

Slide 1 :

1) The inputs to the 17 X 17 Multiplier are two signed numbers, with 17 bits magnitude.

Two types of Multiplication using this Multiplier :

- Pure Integer multiplication of 16 bits inputs (signed) when the adjustment block after multiplication is not required.
 - Multiplication of data of type "MyFloat", ie. 17 bits (Mantissa) magnitude representation and 6 bits (Exponent) to represent the position of the decimal point. In this case the number in the mantissa is adjusted to lie between 0.5 and 0.99, ie the MSB in the mantissa is kept as 1 by adjustment after each multiplication. In "MyFloat" data type the mantissa is kept between 0.5 and 0.99 while the exponent keeps track of the decimal point.
- 2) Load from inside is required when the previous MULT O / P is the input for the next Multiplication.

Description of the previous slides

3) Inputs from a different context come into Reg A and Reg B. Use of Load from outside

4) CSLA Requirements :-

O / P buffered Regs A and B = 3 CSLA

Parallel Multiplier = 37 CSLA

Mult Sign = 1/4 CSLA

4 bit Up counter = 1/2 CSLA.

Buffer for Adjusted mantissa = 1 & 1/4 CSLA

Miscellaneous = 1 CSLA.

Cumulative CSLA Reqs = 43.

SLIDE 2 :

1) 6 bit signed adder for the exponent.

2) Buffer when the exponent comes from the previous M U L T.

3) CSLA Requirements :-

6 bit signed adder = 2 CSLA (max) , Buffer = 1/2 CSLA.

Cumulative CSLA Reqs = 46.

Description of the previous slides

Slide 3 :

- 1) The mantissa and Exponent adjust block after "MyFloat" multiplication.

Blocks and CSLA Requirements :-

17 bit left shift Register = 2 CSLA (max).

Exponent adjust block = 2 CSLA (max).

Miscellaneous = 1 CSLA.

Cumulative CLSA Reqs = 51.

Slide 4:

To convert MyFloat to Integer for accumulation (16 bits fixed point).

5 bit Down Counter = 1 CSLA.

Combinational Logics = 1/4 CSLA.

17 bit shift Register (left / right) = 4 CSLA (max).

Miscellaneous = 1 CSLA.

Cumulative CSLAs = 58 .

Description of the previous slides

Slide 5 :

The accumulator, used for accumulation of the multiplier outputs.

1) CSLA Requirements and Blocks :-

16 bit signed Accumulator = 5 CSLAs.

3 bit down counter = 1/2 CSLA.

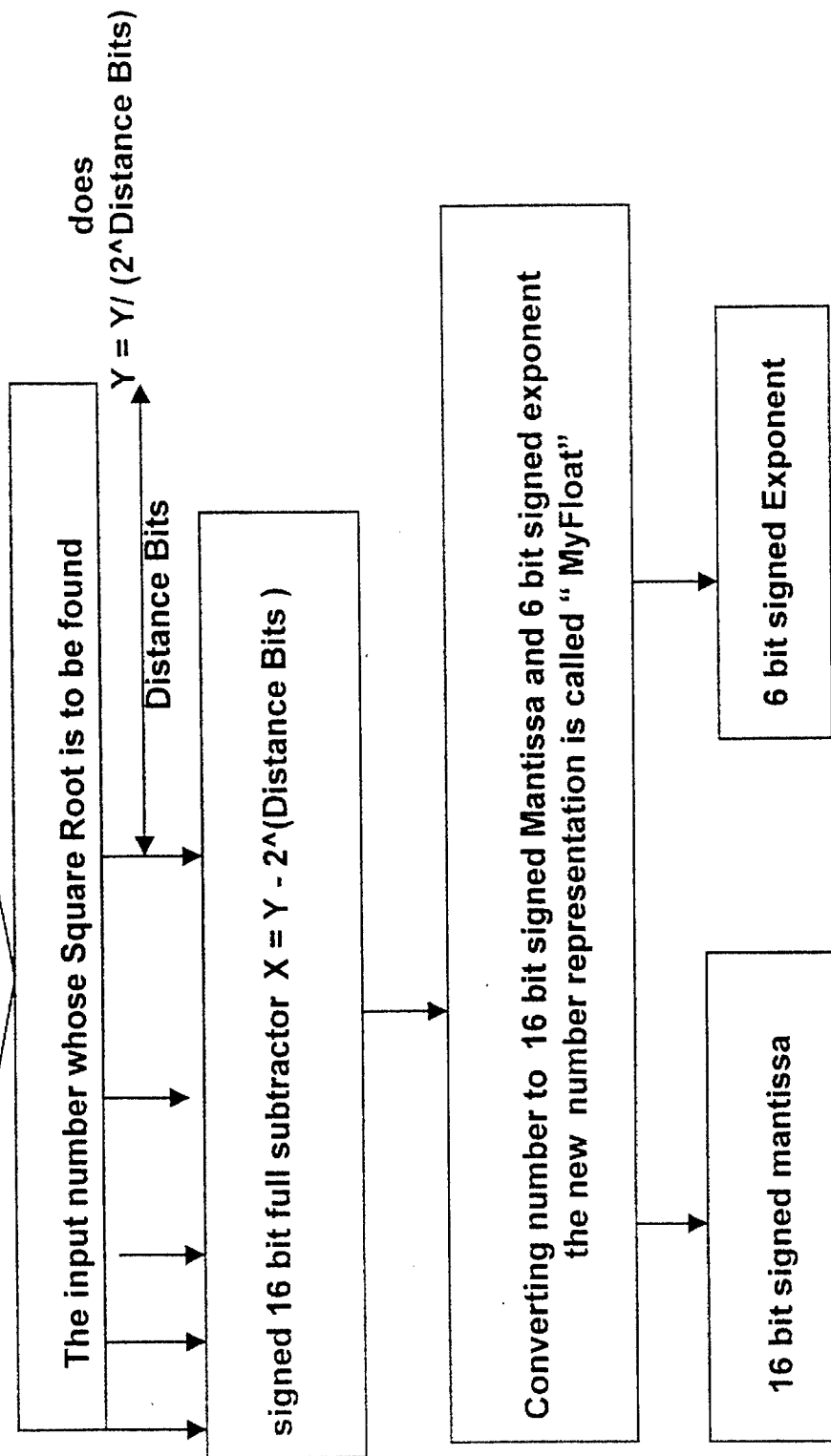
Miscellaneous + OR gate = 1 CSLA.

Cumulative CSLAs = 64.

In case there is a shortage of CSLAs the hardware in slide 5, ie the accumulator can be moved to a partially vacant context (such exists). If such occurs then after each multiplication (+ shifting and adjustment , if required) the results are passed to the accumulator context.

The shifting for the "MyFloat" to Integer conversion may take a varying number cycles depending on the exponent and is done parallelly when the next set of multiplication is being done. This is required when we are calculating the Approximate Square Root (it needs repeated multiplication and accumulation).

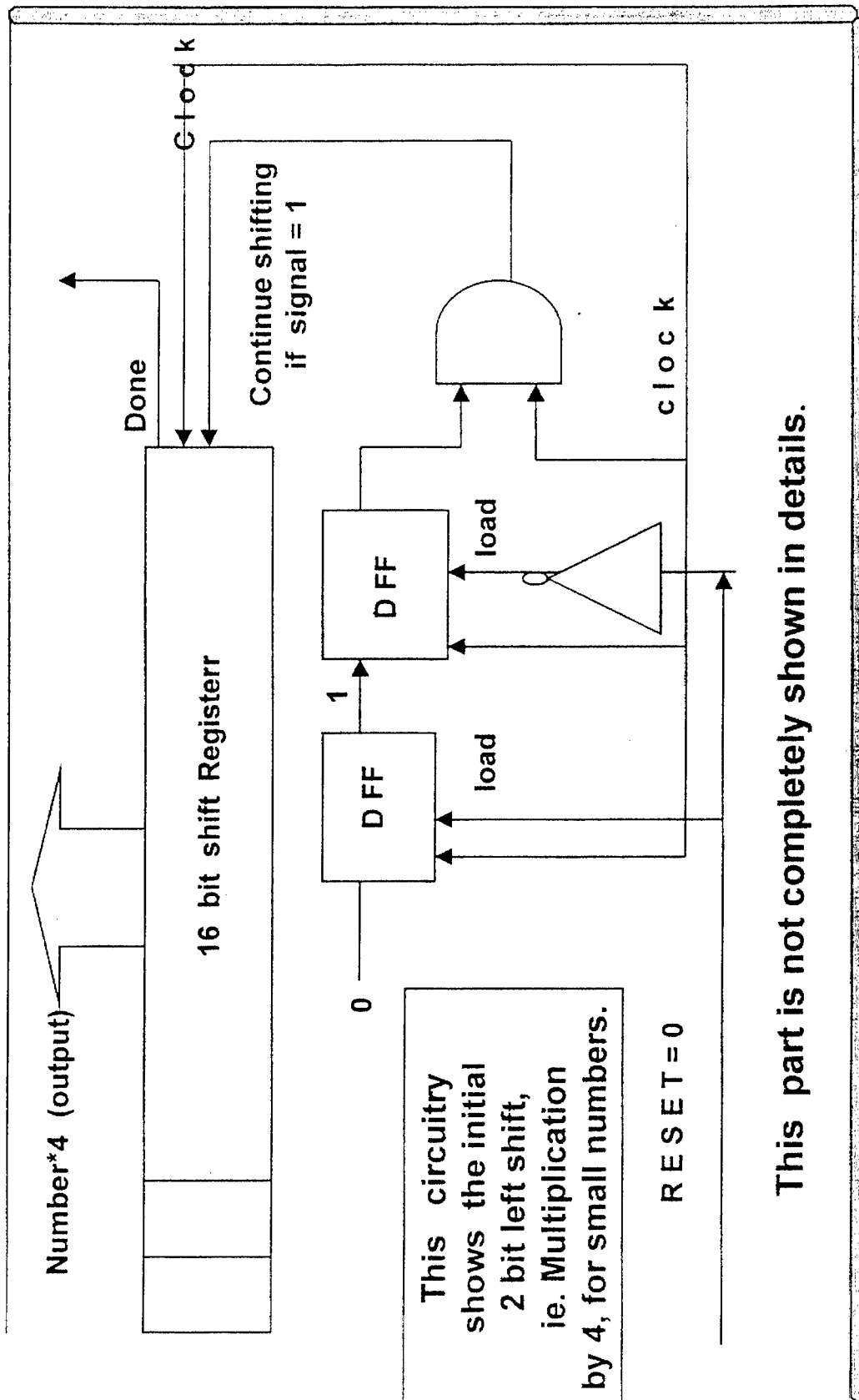
Square Root Front End CSRC 1 cont 4, CSRC 2, cont 1 (slide 6)



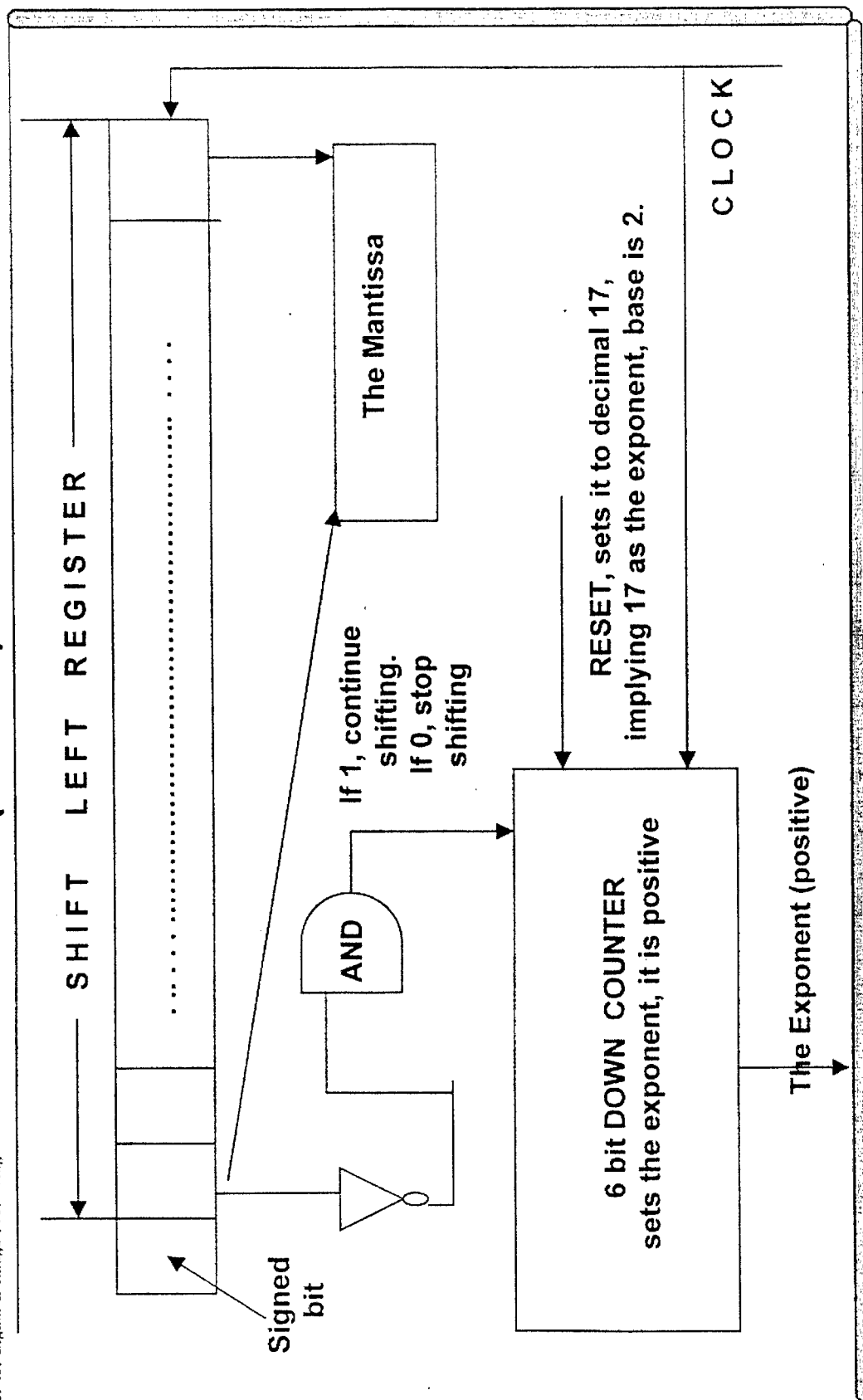
Square Root (Pre-emphasis)

CSRC 1 cont 4, CSRC 2 cont 1

(slide 7)



Integer to MyFloat Converter CSRC 1 cont 4, CSRC 2 cont 1 (slide 8)



Description of the Square Root Front End Slides.

SLIDE 6 :

Blocks and CSLA Requirements :-

Input Buffer , 16 bits = 2 CSLA (max).

Signed 16 bit FULL SUBTRACTOR = 4 CSLA (max).

SLIDE 7:

Square Root Pre-emphasis and O / P de-emphasis.

If the number is small, multiplication of it by an even power of TWO (ie. Even number '2x' of LEFT shifts) and correspondingly dividing the result by HALF of the initial multiplier (ie. Right shifts by 'x'), gives better results. The multiplier depends on the magnitude of the input number whose Square Root we need to find. This is pre-emphasis and de-emphasis.

Approximate CSLA Requirements = 5 CSLAs (max).

Description of the Square Root Front End Slides.

SLIDE 8 :

Integer to “MyFloat” converter :-

16 bit shift Register = 1 CSLA.

6 bit DOWN counter = 2 CSLA (max).

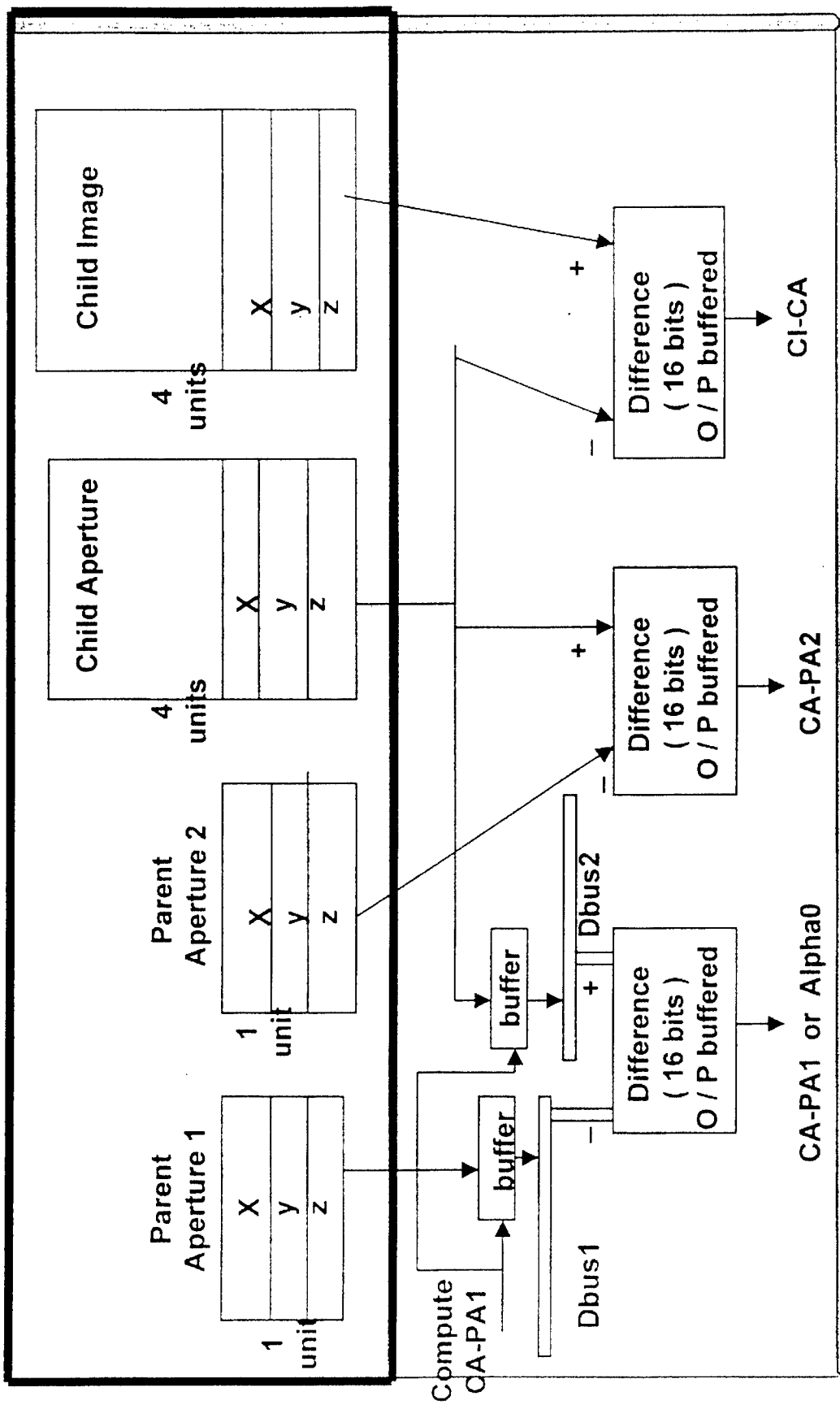
Combinational Logics + Miscellaneous = 4 CSLAs (max).

TOTAL NUMBER OF CSLAs = 18.

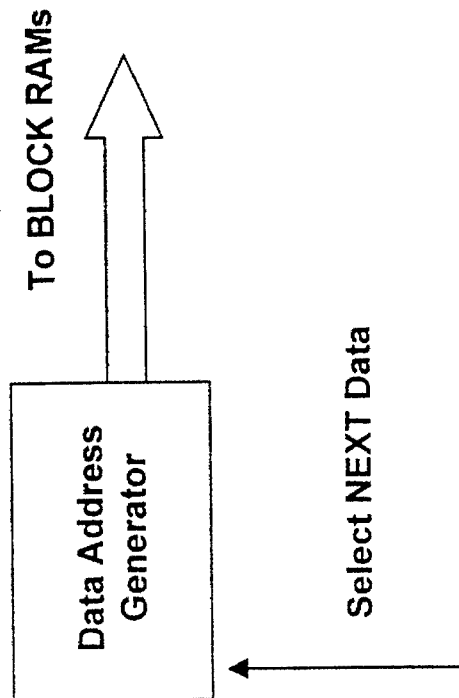
NOTE ON SQUARE ROOT COMPUTATION :-

The Square Root Front End, initially passes the data and the coefficient into the Multiplier Context and then for each iteration it passes the next coefficient into the multiplier when asked by the Mult Context. The resulting Square Root value is gradually accumulated as a 16 bit integer in the accumulator within the multiplier context. The present Square Root Algo requires 8 coefficients, so the multiplier context requests the Square Root context (8 - 1 =) 7 times during the computation.

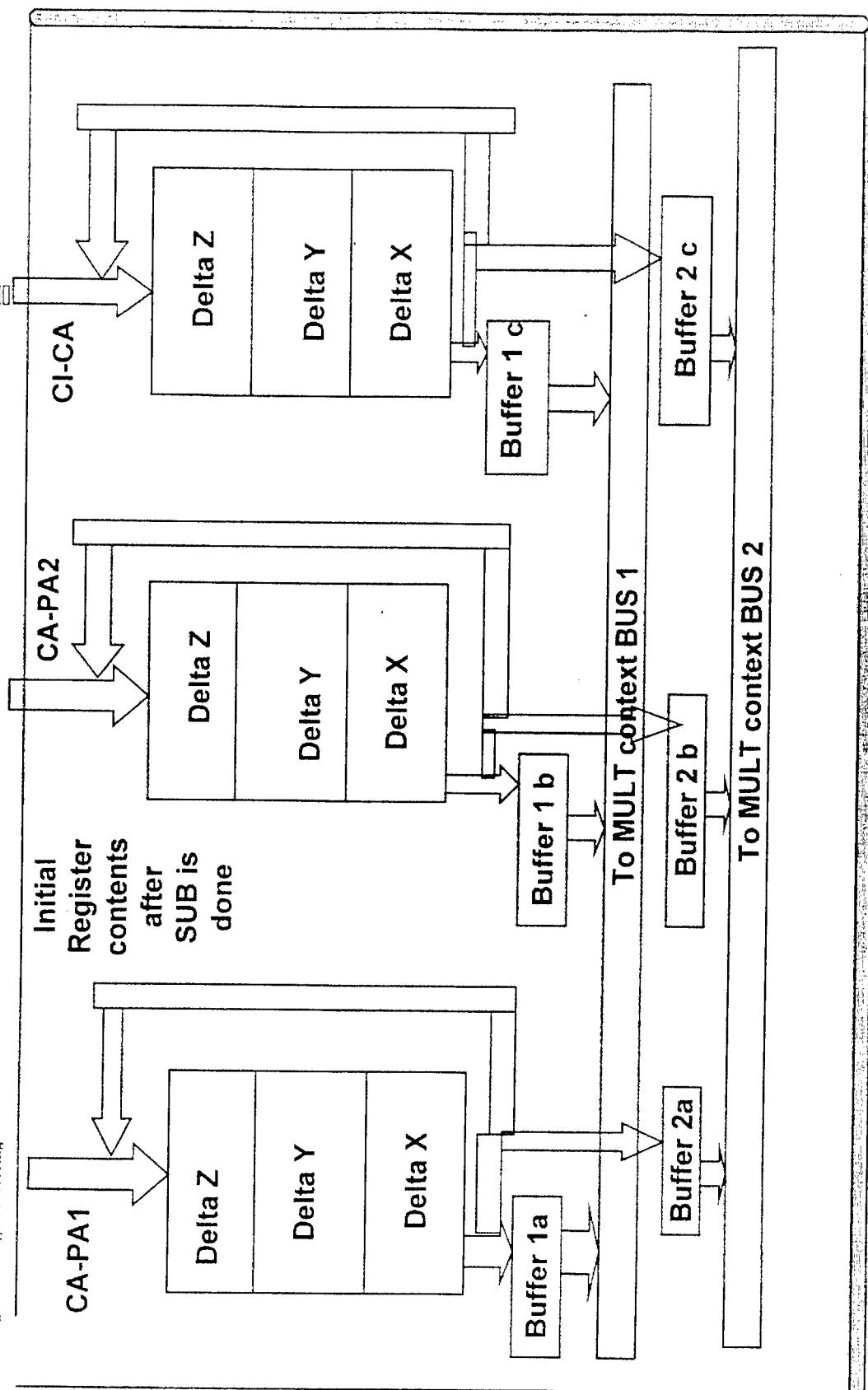
Calculate parent Ranges front end (CSRC 1, context 1) (slide 9)



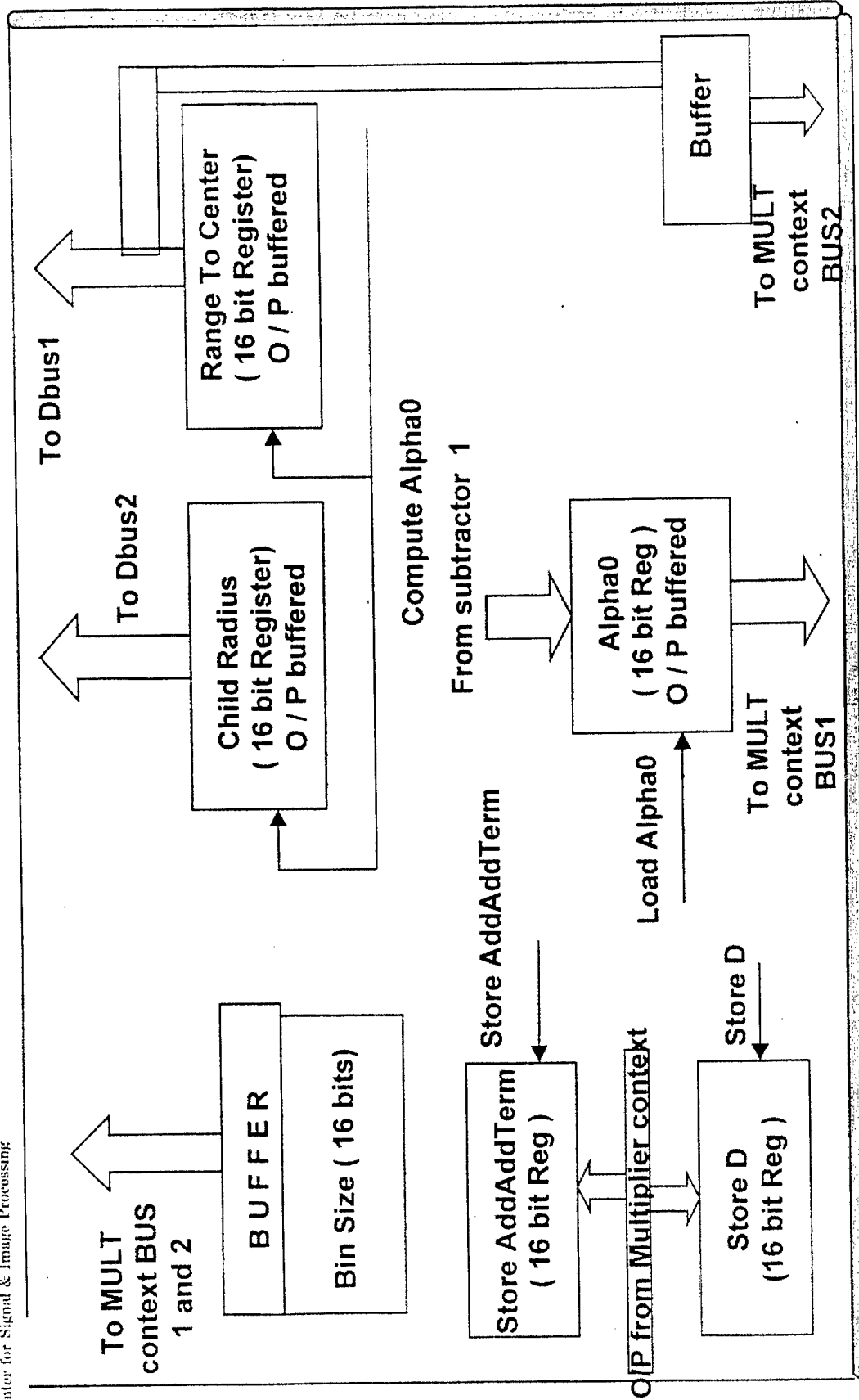
CSRC 1, context 1 (Slide 10)



CSRC 1, context 1. (slide 11)



CSRC 1, context 1. (slide 12)



Description of CalculateParentRanges Front End Context (slides 9 - 12)

SLIDE 9 :

- 1) The portion within the thick black lined box contain input data to be put into the BLOCK RAMs. We need 8 Block RAMs, two for each 16 bit data and four such data sets { parentAperture 1 (PA1) and 2 (PA2), Child Aperture (CA), and Chile Image (CI) Positions}.
- 2) Three 16 bit Subtractor Blocks ($3 * 4 = 12$ CSLAs)
- 3) Two 16 bit Buffers (2 CSLAs).
- 4) Architectural support to reuse one of the subtractor blocks in later calculations.
- 5) The differences, CA-PA1, Alpha0 (calculated later), CA-PA2, CI-CA computed.

Description of CalculateParentRanges Front End Context (slides 9 - 12)

SLIDE 10 :

- 1) Address Generator for selecting the Block RAMs. Use of a 4 bit counter which selects the lower 4 bits (16 locations) in each Block RAM. Idea is to select the data locations that will be used together concurrently.

Example : - Initially find the difference of all the Z locations and store delta Z, followed by Y and X locations to get delta Y and delta X.

No of CSLAs = 1.

SLIDE 11 :

- 1) 3 bit shift Registers, 16 in number (with circular shifting), each for Ca-PA1, CA-PA2, CI-CA. The circular shifting facility is kept so that the registers maintain the same values after providing data for some computation. = 9 CSLAs.

Slides 9 to 12

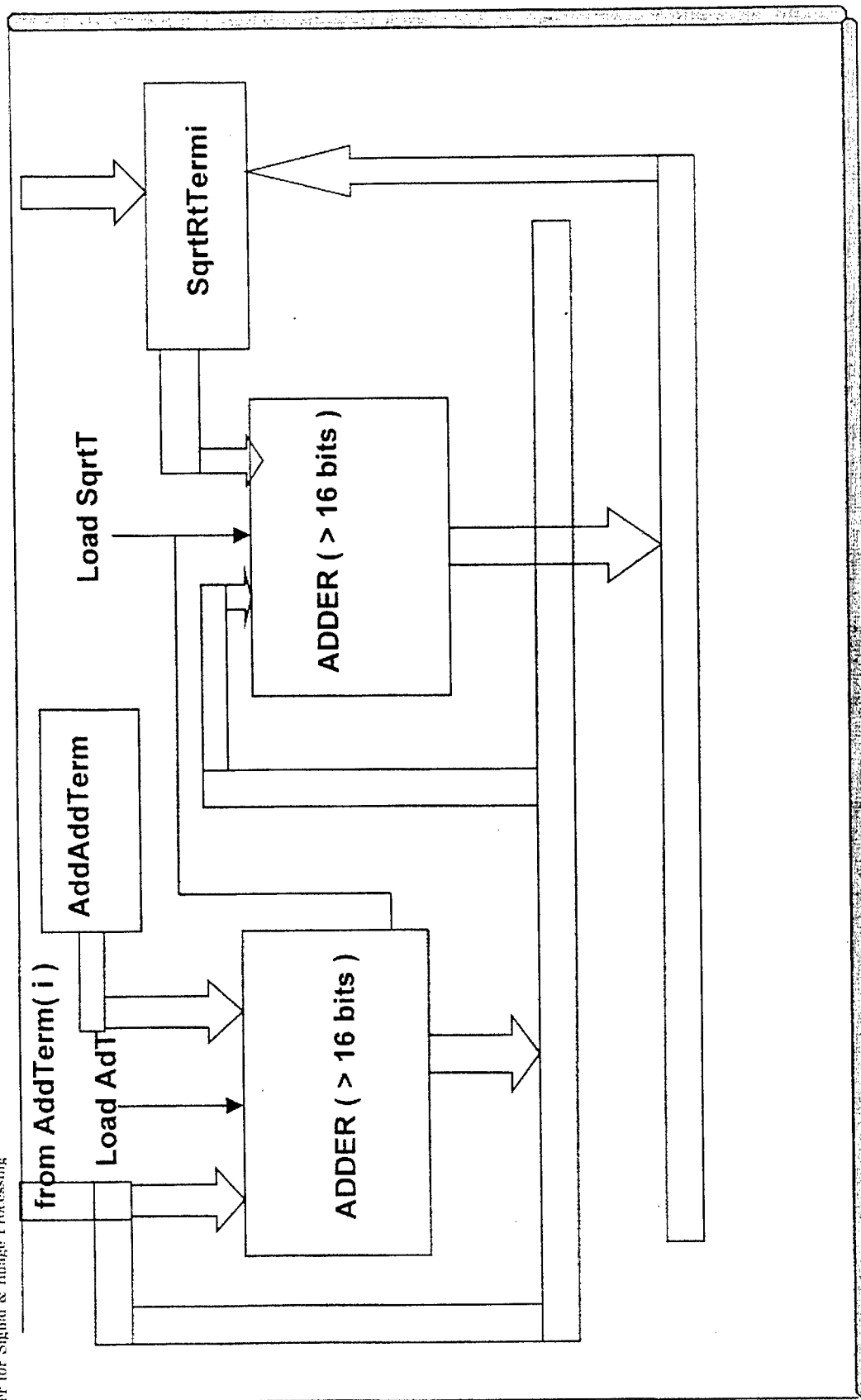
2) 6, 16 bit Buffers = 6 CSLAs. (if associated buffers not available)

SLIDE 12 :

Data Storage Registers = 7 CSLAs.

Total Number of CSLAs in the context = 37 to 40.
CSLA for Control Logics = ???

CSRC1 &2 (context 3) (slide 13)



Description of Slide 13.



2, 16 bit adders = 8 CSLAs.
2, 20 bit buffers = 4 CSLAs.
Control Logics = ???

Timing calculations



Assuming Clock period is 40 nsec and the Read time of the CS ROMs as 10 nsec.

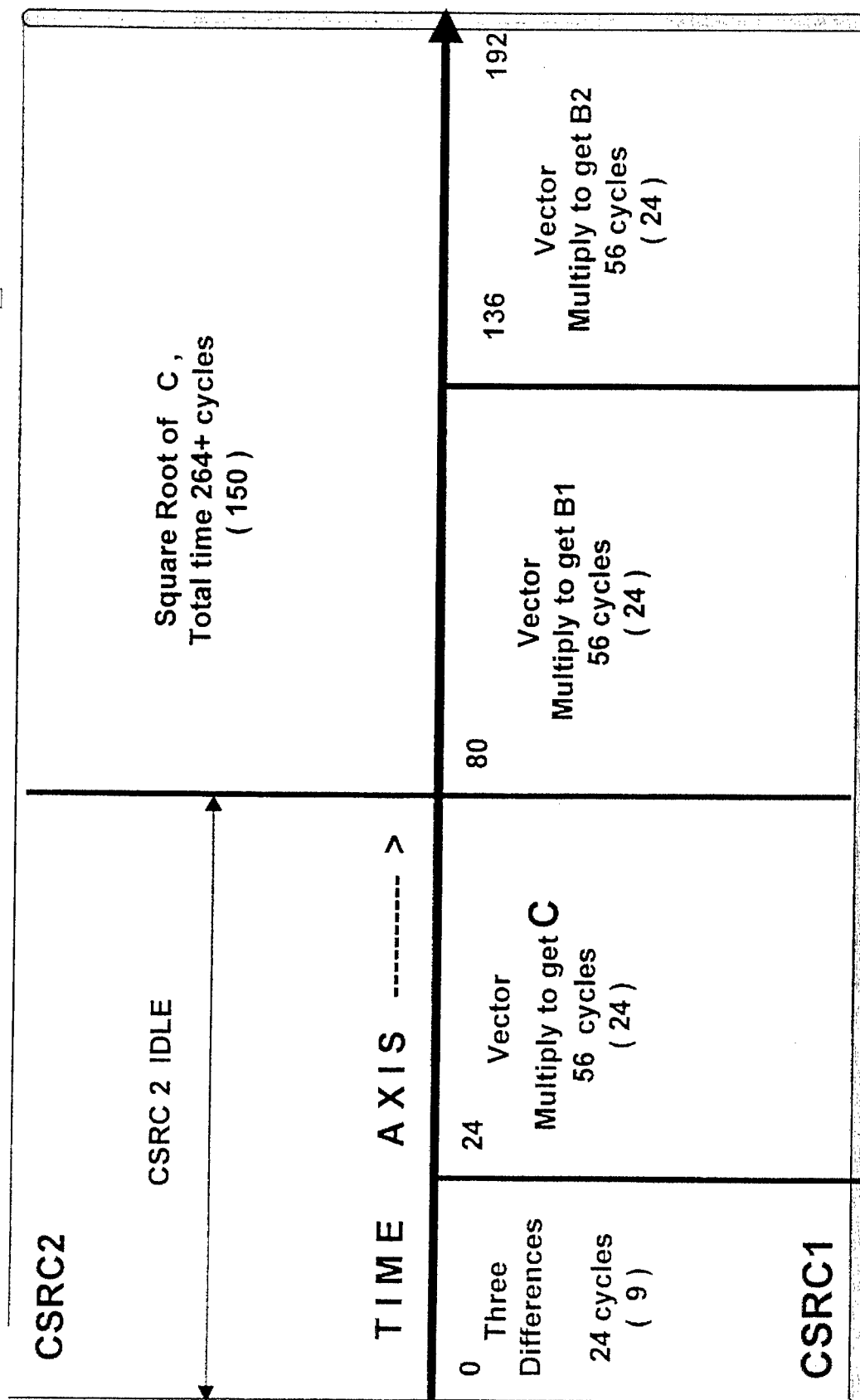
Considering that within one clock period we can have a combinational logic having a maximum of TWO CS ROMs in SERIES.

The timings for the Multiplication and additions may become much less than that shown. The present values give the timings if the carry facility provided by the CSRC is absent.

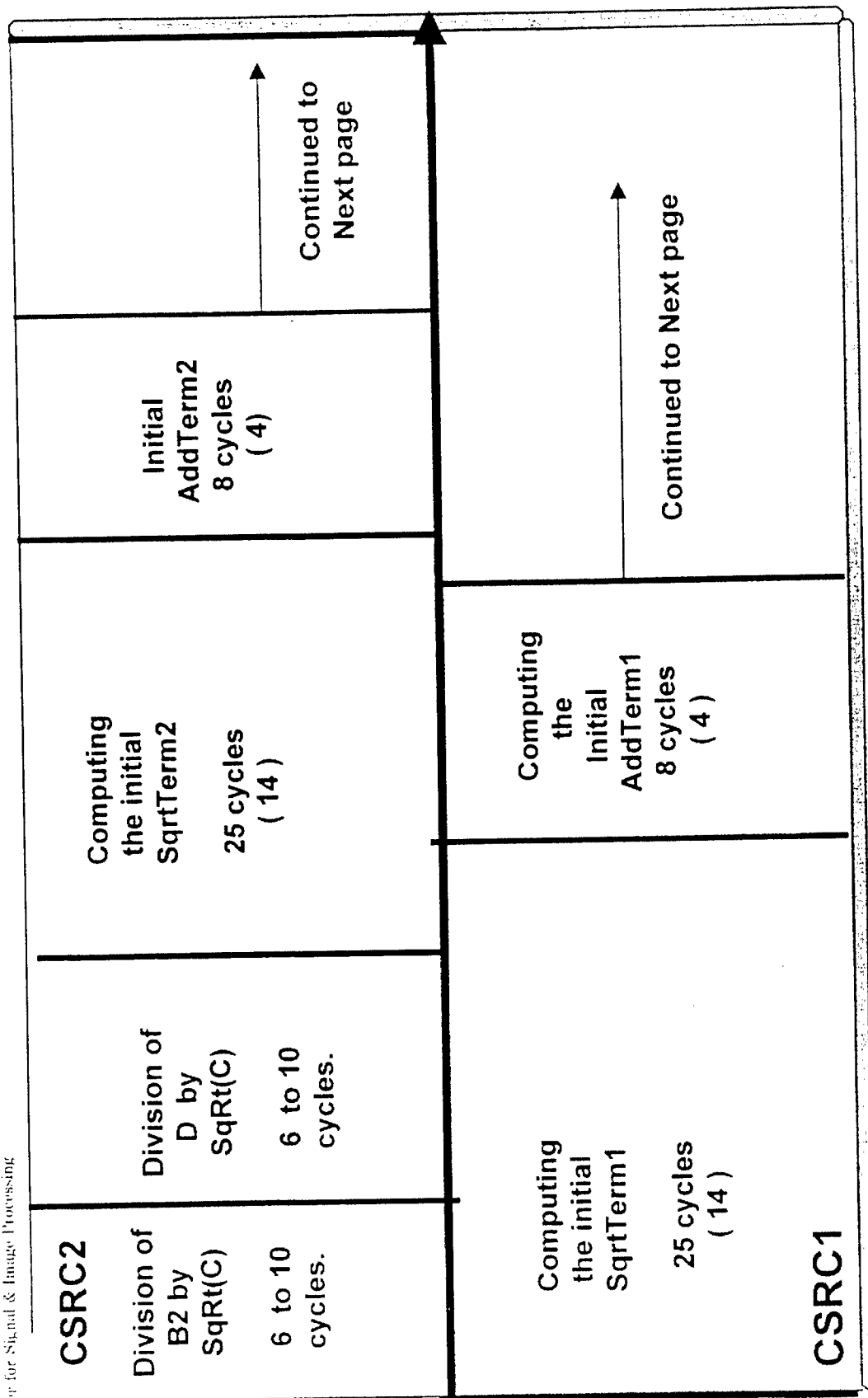
The timings within the brackets may be the times in clock cycles for the better case.

Also, implementation of control Logics within the CSRC may not give a good performance as it is not a Full Custom Design.

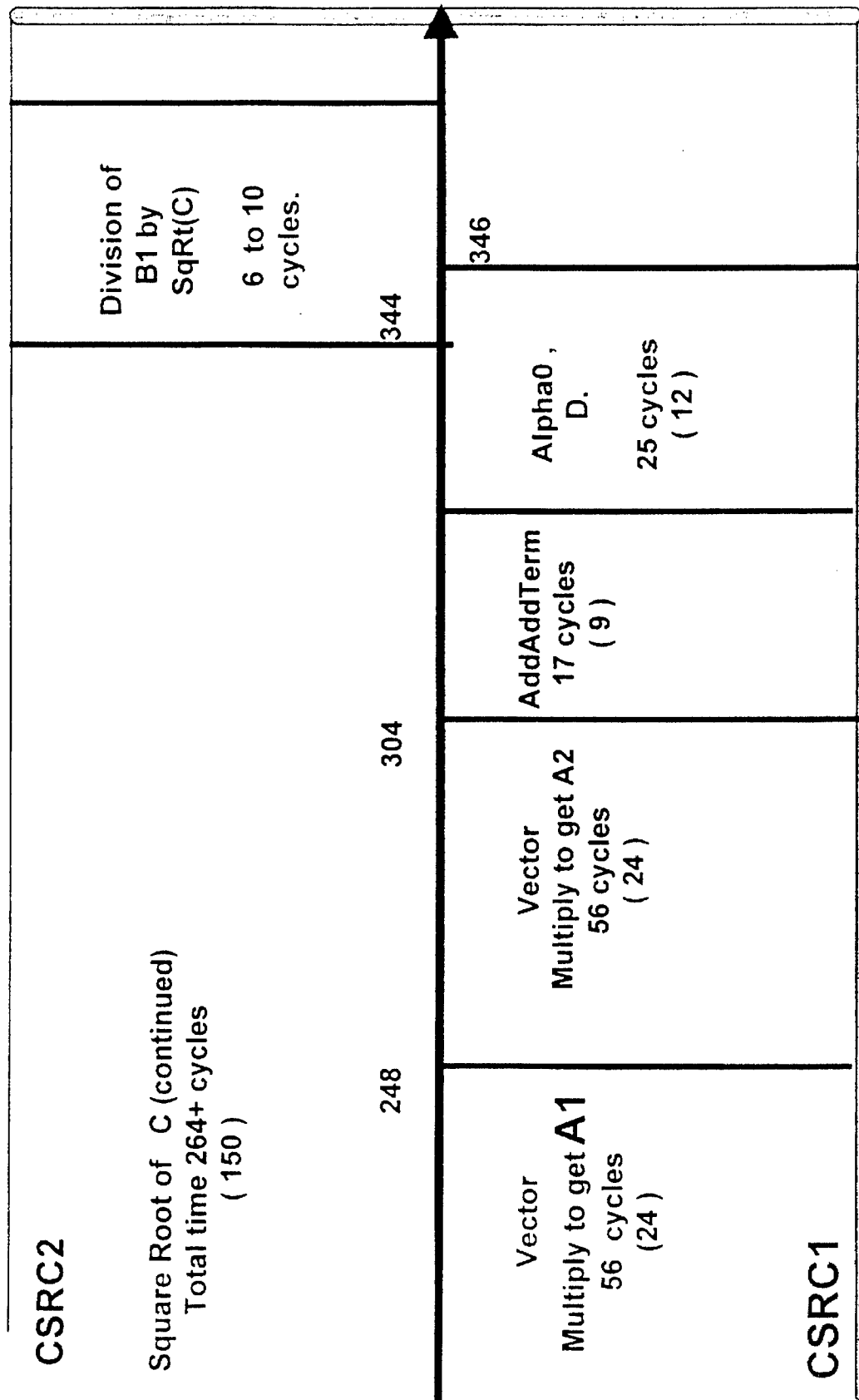
The data flow and Approximate timings(in Clock cycles).



The data flow and Approximate timings(in Clock cycles).



The data flow and Approximate timings(in Clock cycles).



***MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)***

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.